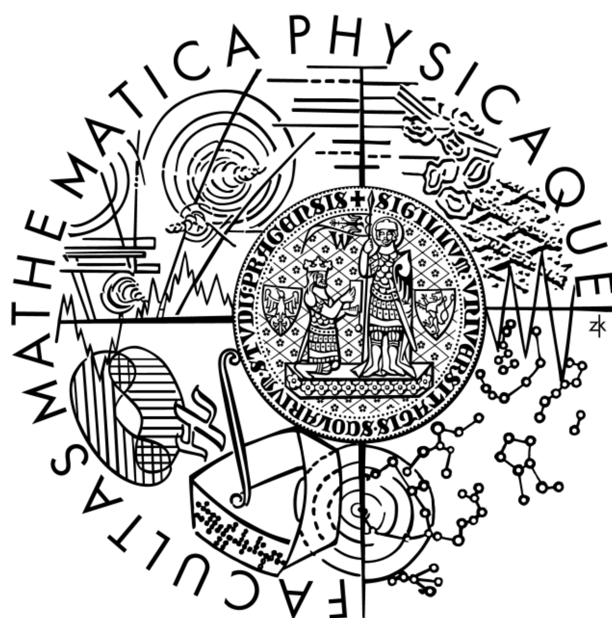
CHARLES UNIVERSITY IN PRAGUE
FACULTY OF MATHEMATICS AND PHYSICS

HABILITATION THESIS



RNDr. PAVEL SURYNEK, PH.D.

COOPERATIVE PATH FINDING FOR MULTIPLE ROBOTS

DEPARTMENT OF THEORETICAL COMPUTER SCIENCE AND
MATHEMATICAL LOGIC

PRAGUE 2015

Table of Contents

Commentary
3

Impact factor journal articles

Solving Abstract Cooperative Path-Finding in Densely Populated Environments
24

On the Complexity of Optimal Parallel Cooperative Path-Finding
85

The Impact of a Bi-connected Graph Decomposition on Solving Cooperative Path-finding Problems
122

Redundancy Elimination in Highly Parallel Solutions of Motion Coordination Problems
138

Pre-processing in Boolean Satisfiability Using Bounded (2,k)-Consistency on Regions with Locally Difficult Constraint Setup
160

Articles in proceedings of leading conferences

Reduced Time-Expansion Graphs for Solving Cooperative Path Finding Sub-optimally
195

An Alternative Eager Encoding of the All-Different Constraint over Bit-Vectors
212

Articles concerning recent research

An Improved Sub-optimal Algorithm for Solving (N^2-1)-Puzzle and Applications in Cooperative Path-Finding
219

Time Expansion Propositional Encodings for Makespan Optimal Solving of Cooperative Path Finding Problem
257

Commentary

This habilitation thesis is a selection of works related to the problem of *cooperative path finding* (CPF)[1,2,3] that I have written since 2009 until the present time. I started to work on the CPF problem (that is also known as *multi-robot path planning*[4] – MRPP, *multi-agent path finding*[5] – MAPF, or *pebble motion on a graph* - PMG) right after finishing my doctoral dissertation. CPF represented a fresh problem at that time for me, very different from my previous research (which was focused on application of constraint programming techniques in domain independent planning), which together gave me strong impetus to achieve new results. CPF was also a fresh problem to artificial intelligence community at that time to certain extent, which constituted highly competitive situation. This situation motivated me to establish an internationally recognized research in the topic.

1. A Combinatorial Approach to Cooperative Path Finding

The problem of CPF takes place in a certain environment where mobile agents are deployed[6]. Each agent starts at a given initial position and it is assigned a goal position where it needs to relocate. The task in CPF is to find a spatial temporal path for each agent so that agents can relocate to their goals cooperatively by fol-

[1] Silver, D.: *Cooperative Pathfinding.* Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2005), pp. 117-122, AAAI Press, 2005.

[2] Yu, J., LaValle, S. M.: *Fast, near-optimal computation for multi-robot path planning on graphs.* The 27th AAAI Conference on Artificial Intelligence (AAAI 2013), Bellevue, WA, USA, late breaking track, AAAI Press, 2013.

[3] Wang, K. C., Botea, A.: *MAPP: a Scalable Multi-Agent Path Planning Algorithm with Tractability and Completeness Guarantees.* Journal of Artificial Intelligence Research (JAIR), Volume 42, pp. 55-90, AAAI Press, 2011.

[4] Ryan, M. R. K. *Exploiting Subgraph Structure in Multi-Robot Path Planning.* Journal of Artificial Intelligence Research (JAIR), Volume 31, pp. 497-542, AAAI Press, 2008.

[5] Sharon, G., Stern, R., Goldenberg, M., Felner, A.: *The increasing cost tree search for op-timal multi-agent pathfinding.* Artificial Intelligence, Volume 195, pp. 470-495, Elsevier, 2013.

[6] Wang, K. C., Botea, A.: *Scalable Multi-Agent Pathfinding on Grid Maps with Tractability and Completeness Guarantees.* Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010), pp. 977-978, Frontiers in Artificial Intelligence and Applications, Volume 215, IOS Press, 2010.

lowing these paths. Cooperation among agents means that agents must not collide with each other and with obstacles while relocating towards their goals. The strong cooperative aspect implies the need to consider all the agents together during path planning and it represents also the major challenge in the problem.

It is assumed, in the version of the problem we studied, that paths are constructed centrally and full observability of the environment and positions of all the agents are granted to the centralized planning mechanism[1]. Agents themselves make no decisions - they merely execute the centrally created plan. A simple instance of the CPF problem where agents need to pass collaboratively through the narrow corridor is shown in Figure 1.

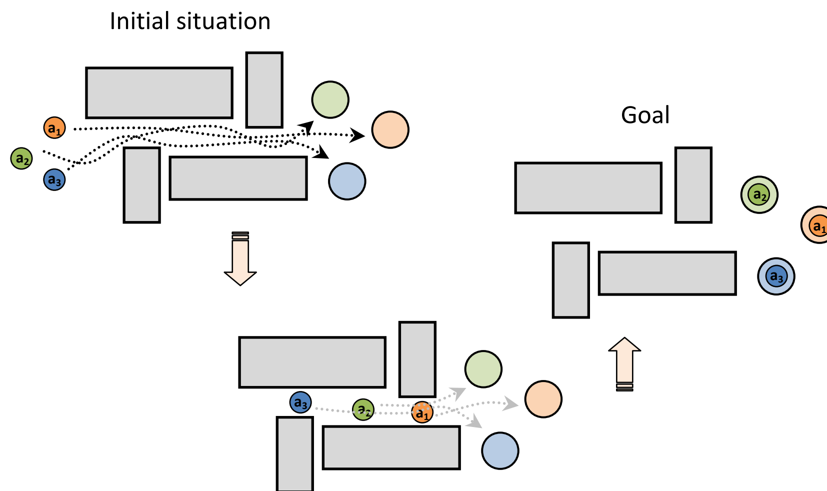


Figure 1. *An example of cooperative path-finding problem (CPF).* Three agents a_1, a_2, and a_3 need to relocate from their initial positions through a narrow corridor represented by obstacles to their goal positions. A process of relocation of agents is illustrated – agents are ordered before entering the corridor to pass through smoothly.

An abstraction in which the environment, where agents are moving, is modeled as an undirected graph is usually adopted[2]. The graph abstraction allows hig-

[1] Wang, K. C. *Bridging the Gap between Centralised and Decentralised Multi-Agent Pathfinding*. Proceedings of the 14th Annual AAAI/SIGART Doctoral Consortium (AAAI-DC 2009), pp. 23-24, AAAI Press, 2009.

[2] Ryan, M. R. K. *Graph Decomposition for Efficient Multi-Robot Path Planning*. Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 2003-2008, IJCAI Conference, 2007.

hlighting the discrete combinatorial nature of the problem. Vertices of the graph represent locations and edges represent passable regions between pairs of locations. Agents are modeled as discrete items placed in vertices of the graph. Space occupancy by agents is modeled by a constraint that at most one agent is placed in each vertex.

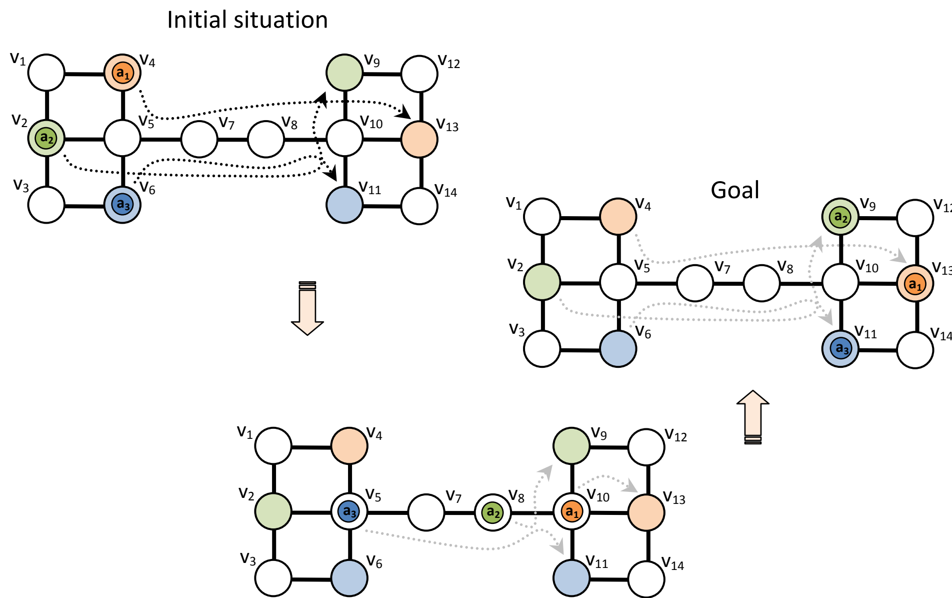


Figure 2. *Cooperative path-finding in an abstract space of undirected graph.* Environment where agents are deployed is modeled as an undirected graph with vertices representing locations and edges representing passable regions. The graph theoretical abstraction of the problem highlights combinatorial aspects of the problem.

An agent can move into neighboring vertex in a single time step assuming that the target vertex is vacant or being vacated and no other agent is trying to enter the same target vertex simultaneously. At least one vertex must be vacant to allow agents to move. Note that various schemes of agent movements exist; sometimes rotations of agents along cycles are allowed with no need to have a vacant vertex inside the cycle[1]. Also note that it is a challenging issue how to sample the real environment in order to build a graph abstraction which correctly reflects the im-

[1] Yu, J., LaValle, S. M. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI 2013), AAAI Press, 2013.

portant properties of the original environment[1] - this issue was however out of scope of the presented research. A simple graph abstraction for the CPF problem from Figure 1 is shown in Figure 2.

Although the graph theoretical abstraction brings a significant simplification to CPF as lot of details are filtered out, the problem remains computationally difficult. It is known that if a solution of the *makespan* (the time or the number of steps needed to execute the solution) that is as small as possible, then the problem is *NP*-hard[2,3]. Due to existence of algorithms capable of generating makespan sub-optimal solutions of the polynomial size with respect to the size of the input[4,5], the decision variant of the problem, in which we ask whether a solution of a given makespan exists, is *NP*-complete.

There are many practical motivations for CPF ranging from unit navigation in computer games[6] to item relocation in automated storage (well known KIVA robots[7] represent a successful commercial application of CPF). Interesting motivations can be also found in traffic where problems like vessel avoidance at sea are of great practical importance[8]. An analogical challenge appears in the air where

[1] Čáp, M., Novák, P., Vokřínek, J., Pěchouček, M. *Multi-agent RRT: sampling-based cooperative pathfinding.* International conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2013), pp. 1263-1264, IFAAMAS, 2013.

[2] Ratner, D., Warmuth, M. K.: *Finding a Shortest Solution for the N × N Extension of the 15-PUZZLE Is Intractable.* Proceedings of the 5th National Conference on Artificial Intelligence (AAAI 1986), pp. 168-172, Morgan Kaufmann, 1986.

[3] Surynek, P. *On the Complexity of Optimal Parallel Cooperative Path-Finding.* Fundamenta Informaticae, Volume 137, Number 4, pp. 517-548, IOS Press, 2015.

[4] Wilson, R. M.: *Graph Puzzles, Homotopy, and the Alternating Group.* Journal of Combinatorial Theory, Ser. B 16, pp. 86-96, Elsevier, 1974.

[5] Surynek, P.: *Solving Abstract Cooperative Path-Finding in Densely Populated Environments.* Computational Intelligence (COIN), Volume 30, Issue 2, pp. 402-450, Wiley, 2014.

[6] Sturtevant, N. R. *Benchmarks for Grid-Based Pathfinding.* IEEE Transactions on Computational Intelligence and AI in Games, Volume 4(2), pp. 144-148, IEEE Press, 2012.

[7] KIVA Systems. *Official web site.* http://www.kivasystems.com/, 2015 [accessed in April 2015].

[8] Kim, D., Hirayama, K., Park, G.-K. *Collision Avoidance in Multiple-Ship Situations by Distributed Local Search.* Journal of Advanced Computational Intelligence and Intelligent Informatics (JACIII), Volume 18(5), pp. 839-848, Fujipress, 2014.

availability of air-drones implies a need for developing cooperative air-traffic control mechanisms[1].

2. Overview of Selected Contributions to Cooperative Path Finding

Following works by the author of this thesis are included for presentation. The selection of works was mostly guided by the leading role of the author in their creation and by relevance to the topic of CPF. Five of the selected works were published in impact factor journals and two in proceedings of leading artificial intelligence conferences. Two more works have not yet been published and were included into the appendix to provide more complex figure about the recent research of the author in CPF.

Impact factor journal articles:

(i) Pavel **Surynek**: *Solving Abstract Cooperative Path-Finding in Densely Populated Environments.* Computational Intelligence (COIN), Volume 30, Issue 2, pp. 402-450, Wiley, 2014.

(ii) Pavel **Surynek**: *On the Complexity of Optimal Parallel Cooperative Path-Finding.* Fundamenta Informaticae, Volume 137, Number 4, pp. 517-548, IOS Press, 2015.

(iii) Pavel **Surynek**, Petra **Surynková**, Miloš **Chromý**: *The Impact of a Bi-connected Graph Decomposition on Solving Cooperative Path-finding Problems.* Fundamenta Informaticae, Volume 135 (3), pp. 295-308, IOS Press, 2014.

(iv) Pavel **Surynek**: *Redundancy Elimination in Highly Parallel Solutions of Motion Coordination Problems.* International Journal on Artificial Intelligence Tools (IJAIT), Volume 22, Number 05 (19 pages), World Scientific, 2013, ISSN 0218-2130.

(v) Pavel **Surynek**: *Pre-processing in Boolean Satisfiability Using Bounded (2,k)-Consistency on Regions with Locally Difficult Constraint Setup.* International Journal on Artificial Intelligence Tools (IJAIT), Volume 23, Number 01 (29 pages), World Scientific, 2014, ISSN 0218-2130.

[1] Michael, N., Fink, J., Kumar, V. *Cooperative manipulation and transportation with aerial robots.* Autonomous Robots, Volume 30 (1), pp. 73-86, Springer, 2011.

Articles in proceedings of leading conferences:

(vi) Pavel **Surynek**: *Reduced Time-Expansion Graphs for Solving Cooperative Path Finding Sub-optimally.* Proceedings of the 24rd International Joint Conference on Artificial Intelligence (IJCAI 2015), Buenos Aires, Argentina, IJCAI/AAAI Press, 2015.

(vii) Pavel **Surynek**. *An Alternative Eager Encoding of the All-Different Constraint over Bit-Vectors.* Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012), pp. 927-928, Montpellier, France, IOS Press, 2012, ISBN 978-1-61499-097-0.

Articles concerning recent research:

(viii) Pavel **Surynek**, Petr **Michalík**: *An Improved Sub-optimal Algorithm for Solving (N^2-1)-Puzzle and Applications in Cooperative Path-Finding*, 2015.

(ix) Pavel **Surynek**: *Time Expansion Propositional Encodings for Makespan Optimal Solving of Cooperative Path Finding Problem*, 2015.

Article (i)

The first presented article (i) is devoted to a detailed design and analysis of polynomial time sub-optimal algorithms for solving CPF over *bi-connected graphs*[1] - algorithms are called BIBOX and BIBOX-θ. Both algorithms have been initially published in workshop and conference proceedings[2,3,4]. The original article[5] refer-

[1] Westbrook, J., Tarjan, R. E. *Maintaining bridge-connected and biconnected components on-line.* Algorithmica, Volume 7, Number 5&6, pp. 433–464, Springer, 1992.

[2] Surynek, P.: *Domain-Dependent View of Multiple Robots Path Planning.* Proceedings of the 4th European Starting AI Researcher Symposium (STAIRS 2008), Patras, Greece, pp. 175-186, IOS Press, 2008

[3] Surynek, P.: *Finding Plans for Rearranging Robots in θ-like Environments.* Proceedings of the 27th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2008), Edinburgh, United Kingdom, pp. 128-129, Heriot-Watt University, 2008.

[4] Surynek, P.: *An Application of Pebble Motion on Graphs to Abstract Multi-robot Path Planning.* Proceedings of the 21st International Conference on Tools with Artificial Intelligence (ICTAI 2009), Newark, NJ, USA, pp. 151-158, IEEE Press, 2009.

[5] Surynek, P.: *A Novel Approach to Path Planning for Multiple Robots in Bi-connected Graphs.* Proceedings of the 2009 IEEE International Conference on Robotics and Automation (ICRA 2009), Kobe, Japan, pp. 3613-3619, IEEE Press, 2009.

ring about the BIBOX algorithm is so far the most cited work of the author. One of the most valuable citations is an in-depth reference about the BIBOX algorithm within a tutorial[1] presented to general audience at AAAI 2013.

The BIBOX algorithm takes an input bi-connected graph other than the cycle and constructs its *decomposition into ears*[2] (it holds that any bi-connected can be constructed by incremental adding of ears to a currently constructed graph while a cycle is taken as the initial graph). The algorithm then proceeds according to the ear decomposition by arranging agents to their goal positions in the last ear of the ear decomposition (the ordering of ears is determined by their position in the construction sequence – the initial cycle is considered as first). Agents are arranged into the ear in the stack like manner, that is, agents are pushed into the ear by rotating it after the next agent arrives to the entrance of the ear. The final push operation (the final rotation) shifts agents to their respective goal positions within the ear.

After the placement of agents in the ear is finished, the ear can be ruled out from further consideration, that is, the task of agent placing reduces to a smaller bi-connected graph where the same process is applied inductively until the initial cycle of the ear decomposition is reached. A special process is used to arrange agents into the initial cycle of the decomposition. Placing agents into regular ears of the ear decomposition requires only one unoccupied vertex while two unoccupied vertices are required to arrange agents into the initial cycle to be able to generate all the permutations of agents in the cycle by swaps.

The algorithm runs in $\mathcal{O}(|V|^3)$ for given input graph $G = (V, E)$ and the size of generated solution (that is, the number of moves) is $\mathcal{O}(|V|^3)$ as well.

The BIBOX-θ algorithm improves the solving process of BIBOX over the initial cycle and requires only one unoccupied vertex to be able to arrange agents there. The algorithm uses a *pattern database*, which contains solutions of sub-problems representing swaps of pairs and rotations of triples of agents. If the instance is solvable, the resulting solution can be composed of solutions to these sub-problems stored in the pattern database.

Solutions generated by BIBOX and BIBOX-θ consist of order of magnitude fewer moves than solutions generated by at that time existing comparable algo-

[1] ter Mors, A.: *Moving Agents in a Graph.* The Tutorial Forum of the Twenty-Seventh AAAI Conference on Artificial Intelligence (AAAI-2013), 2013.

[2] Tarjan, R. E.: *Depth-First Search and Linear Graph Algorithms.* SIAM Journal on Computing, Volume 1 (2), pp. 146-160, Society for Industrial and Applied Mathematics, 1972.

rithm[1] albeit the theoretical asymptotic size of solutions is the same. Search-based algorithms for CPF share a common drawback that they do not scale up for larger number of agents while BIBOX algorithms do not care about the number of agents in the environment at all (multiple unoccupied vertices can be utilized by taking the nearest unoccupied vertex to the location where it is needed). Another important advantage of BIBOX algorithms is that they do not rely on necessity to undo some of generated movements as it is in the case later algorithms such as PUSH-AND-SWAP[2] and PUSH-AND-ROTATE[3]. This property enabled to reuse the core idea of the BIBOX algorithm in the design of a new algorithm for solving CPF in unidirectional environments, namely over *strongly bi-connected directed graphs*[4] - the algorithm is called diBOX[5].

As noticed in [6], works on BIBOX algorithms are first that put in relation the algebraic approach to pebble motion on graphs and cooperative path finding. Until that time, only search-based algorithms without completeness guarantees applicable for few agents only were considered for CPF.

[1] Kornhauser, D., Miller, G. L., and Spirakis, P. G.: *Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications.* Proceedings of the 25th Annual Symposium on Foundations of Computer Science (FOCS 1984), pp. 241-250, IEEE Press, 1984.

[2] Luna, R., Berkis, K., E. *Push-and-Swap: Fast Cooperative Path-Finding with Completeness Guarantees.* Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011), pp. 294-300, IJCAI/AAAI Press, 2011.

[3] de Wilde, B., ter Mors, A. W., Witteveen, C.*: Push and Rotate: a Complete Multi-agent Pathfinding Algorithm.* Journal of Artificial Intelligence Research (JAIR), Volume 51, pp. 443-492, AAAI Press, 2014.

[4] Wu, Z., Grumbach, S.: *Feasibility of motion planning on acyclic and strongly connected directed graphs.* Discrete Applied Mathematics, Volume 158(9), pp. 1017 – 1028, Elsevier, 2010.

[5] Botea, A., Surynek, P.: *Multi-Agent Path Finding on Biconnected Directed Graphs.* Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015), Austin, TX, USA, pp. 2024-2030, AAAI Press, 2015.

[6] Röger, G., Helmert, M.: *Non-Optimal Multi-Agent Pathfinding is Solved (Since 1984).* Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS 2012), AAAI Press, 2012.

Article (ii)

The second presented article (ii) concentrates on complexity issues of the problem of finding *makespan optimal* solution of the version of CPF where movements of agents into vertices being simultaneously vacated are allowed. Only the leading agent of the chain of moving agents needs to move into unoccupied vertex while other agents follow it (agents move like a train). Makespan is the number of time steps necessary to execute the CPF solution while parallel moves are possible (note that makespan optimal solution does not need to be optimal in terms of the total number of moves). It has been shown that finding makespan optimal solution is *NP*-hard and the decision version of the problem is *NP*-complete (a question whether there is a solution of the given makespan).

A reduction of propositional satisfiability (SAT)[1,2] to makespan optimal CPF solving is used to show *NP*-hardness in the presented work. Membership of the problem into *NP* class is justified by previous algorithms generating sub-optimal solutions of polynomial size (such as BIBOX and related).

A CPF instance, in which passing through certain vertices by agents simulates assigning of truth-values to propositional variables of the input propositional formula in *conjunctive normal form* (CNF – which is a conjunction of *clauses* where a clause is a disjunction of *literals*, and a literal is either a propositional variable or its negation), is constructed. Several techniques were developed to force agents to move in desired way in makespan optimal solutions. A so-called *vertex locking* technique is used to allow entering a selected vertex at selected time steps only. Vertices in the target CPF instance correspond to occurrences of literals of in the input formula. Thus, in order to simulate truth-value assignments correctly in case when literal corresponding to a single propositional variable has multiple occurrences, propositional consistency needs to be preserved. That is, a group of agents needs to pass either a set of vertices corresponding to positive literals of the given variable or a set of vertices corresponding to negative literals. A special technique, that forces a group of agents to move together (separation between positive and negative literals is not allowed), has been developed.

An interesting property of the reduction of SAT to CPF is that the resulting CPF instance is sparsely occupied by agents (the portion of unoccupied vertices is

[1] Cook, S. A.: *The Complexity of Theorem Proving Procedures.* Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971), pp. 151-158, ACM Press, 1971.

[2] Biere, A., Heule, M., van Maaren, H., Walsh, T.: *Handbook of Satisfiability.* IOS Press, 2009.

proportionally larger than the number of agents). This means that difficult instances of CPF are not only those densely occupied by agents as reduction of hard SAT instances[1] may lead to sparsely occupied CPF instance.

Article (iii)

The important parameter of the BIBOX algorithm is the decomposition of the input bi-connected graph into sequence of ears. Although the theoretical asymptotic estimation of the size of solution and the runtime is independent of the ear decomposition, the practical size of solution and runtime may differ.

The runtime of the BIBOX algorithm for several common ear decompositions is evaluated theoretically and experimentally in the presented article (note that presented results apply for the size of solutions with minor modifications only). Ear decompositions such as that with ears of constant size, linearly increasing size, or quadratic size were evaluated.

The outcome of theoretical and experimental evaluation is that ear decomposition with ears of constant size results in fastest runtime of the BIBOX algorithm.

This article has implications for the new diBOX for which works in almost the same way as BIBOX. It is expectable that short ears (constant size) should be preferred in ear decomposition for the diBOX algorithm. However, detailed experimental evaluation need to be done to confirm this claim. Another question that arises in connection with this article is how to find ear decompositions of required properties[2].

Article (iv)

This article focuses on improving existing sub-optimal of CPF solutions by eliminating certain *redundancies* from them. The important prerequisite of this article was supervising of a software project of author's student[3] who developed soft-

[1] Hoos, H. H., Stützle, T.: *SATLib: An Online Resource for Research on SAT.* Proceedings of Theory and Applications of Satisfiability Testing, 4th International Conference (SAT 2000), pp.283-292, IOS Press, 2000, http://www.satlib.org, [May 2015].

[2] Szigeti, Z.: *On Optimal Ear-Decompositions of Graphs.* Integer Programming and Combinatorial Optimization, 7th International IPCO Conference (IPCO 1999), pp. 415-428, 1999.

[3] Koupý, P.: *GraphRec - a visualization tool for entity movement on graph.* Student project web page, 2011, http://www.koupy.net/graphrec.php, [accessed May 2015].

ware for visualizing solutions of CPFs – the software tool is called GRAPHREC[1,2]. GRAPHREC enables to draw the input graph modeling the environment of CPF in plane automatically. Movements of agents are subsequently animated over the drawn graph. Various graphical tools were implemented in order to make the visual analysis of the solution as comfortable as possible.

Several inefficiencies were observed in solutions generated by BIBOX algorithms using GRAPHREC especially in cases with multiple unoccupied vertices. These inefficiencies were formalized and simple algorithms (polynomial time) for their elimination were suggested in this article.

Simply detectable inefficiencies concerns single agent and has a local nature[3]. For example, a sequence of moves of a single agent is called *redundant* if the agent starts and finishes the sequence of moves at the same vertex in the graph without interfering with other agents. In such a case, the sequence of redundant moves can be eliminated from the solution and a new solution that contains fewer moves and that may be even shorter in terms of makespan is obtained.

Attempts to extend the notion of inefficiency or redundancy of movements from single agent to multiple agents led to idea to replace entire sub-solutions of the input solution where all the agents are considered together with sub-solution of shorter makespan. It has been suggested to replace a sub-solution of original makespan sub-optimal solution with makespan optimal sub-solution. It was expected that sub-solution to be replaced is short, hence although it is *NP*-hard to find makespan optimal solution of CPF, it should be practically feasible in small/easy CPF instances at least.

[1] Surynek, P., Koupý, P.: *Improving Solutions of Problems of Motion on Graphs by Redundancy Elimination.* Proceedings of the ECAI 2010 Workshop on Spatio-Temporal Dynamics (ECAI STeDy 2010), Lisbon, Portugal, pp. 37-42, SFB/TR 8 Reports, Universität Bremen, Germany, 2010.

[2] Surynek, P., Koupý , P.: *Vizualizace jako prostředek k získání znalostí o kvalitě řešení problémů pohybu po grafu (Visualization as a tool for acquiring knowledge about the quality of solutions of problems of motion on graphs).* Proceedings of the Conference Znalosti 2010, Faculty of Management, VŠE Prague, Jindřichův Hradec, Czech Republic, pp. 129-141, Nakladatelství Oeconomica Jindřichův Hradec, 2010, in Czech.

[3] Surynek, P.: *Redundancy Elimination in Highly Parallel Solutions of Motion Coordination Problems.* Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2011), Boca Raton, FL, USA, pp. 701-708, IEEE Press, 2011.

The makespan optimal replacement sub-solution is searched by reducing the problem to propositional satisfiability[1] (SAT). Reductions of CPF to SAT represent a broad topic itself to which one of the following presented articles is devoted. The interesting aspect here is how to choose sub-solutions to be replaced. The approach presented in the article is an *anytime algorithm* called COBOPT (it can be terminated anytime as the algorithm has a valid solution at hand anytime) that gradually increases the length of considered sub-solutions, which may eventually lead to optimization of the entire sub-optimal solution if sufficient time is provided.

Article (v)

SAT instances (propositional formulae), which are result of reductions of instances of a certain problem share common properties that, if well utilized, can help the SAT solver to solve them. This observation independently started research of the author in special consistencies in SAT that can simplify the instance with respect to practical solving effort[2,3]. Although this was initially a research independent of research in CPF, it turned out that it has great applicability in CPF solving as one of the major research directions in CPF solving followed by the author became its reductions to SAT.

The consistency described in the article views SAT as a *constraint satisfaction problem*[4] (CSP) which is a step towards final interpreting the formula as a graph where vertices are represented by literals and edges are represented by conflicts between literals (that is, a pair of literals in conflict cannot take the value $TRUE$ simultaneously). A *clique* in this conflict graph means that at most one literal involved in the clique can be selected to be assigned $TRUE$. In its very basic va-

[1] Kautz, H., Selman, B. *Unifying SAT-based and Graph-based Planning*. Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 1999), pp. 318-325, Morgan Kaufmann, 1999.

[2] Surynek, P.: *An Adaptation of Path Consistency for Boolean Satisfiability: a Theoretical View of the Concept.* Proceedings of the Annual ERCIM Workshop on Constraint Solving and Constraint Logic Programming, 2010 (CSCLP 2010), Berlin, Germany, pp. 16-30, Fraunhofer FIRST, 2010.

[3] Surynek, P.: *Between Path-Consistency and Higher Order Consistencies in Boolean Satisfiability.* Proceedings of the Annual ERCIM Workshop on Constraint Solving and Constraint Logic Programming, 2011 (CSCLP 2011), York, United Kingdom, pp. 120-134, University of York, 2011.

[4] Dechter, R.: *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

riant, the suggested $(2, k)$-*consistency* can be used to detect that we cannot select enough literals to achieve certain goal. This reasoning can be easily illustrated on the well known *pigeon hole principle*[1,2] (n pigeos need to placed into $n-1$ holes so that no two pigeons appear in the same hole) which when interpreted using graphs of conflicts results in several cliques. As it is known that from each clique at most one literal can be selected, it can be easily concluded that there is simply not enough cliques in the graph to satisfy the input instance.

The new consistency not only determines that the input instance is unsolvable but also rules out values and pairs of values (literals) that cannot appear in any solution. To further increase chance of the consistency to make non-trivial inferences, it concentrates on regions of the instance (conflict graph) which has similar statistical patterns as the model of pigeon hole principle.

The $(2, k)$-*consistency* has been implemented as a SAT preprocessing tool called PREPROCESSSIGMA. Experimental evaluation indicated that the new preprocessing tool has been especially successful on instances modeling *integer factorization problem*[3] where it outperformed comparable preprocessing tools such as LIVER, NIVER[4], and HYPRE[5].

[1] Aloul, F. A., Ramani, A., Markov, I. L., Sakallah, K. A.: *Solving Difficult SAT Instances in the Presence of Symmetry.* Proceedings of the 39th Design Automation Conference (DAC 2002), pp. 731-736, USA, ACM Press, 2002, http://www.aloul.net/benchmarks.html, [accessed May 2015].

[2] Aloul, F. Markov, I. L., Sakallah, K.: *Shatter: Efficient Symmetry-Breaking for Propositional Satisfiability.* Proceedings of the Design Automation Conference (DAC 2003), pp. 836-839, ACM Press, 2003, http://www.aloul.net/Tools/shatter/, [accessed May 2015].

[3] Aloul, F. A.: *SAT Benchmarks, Difficult Integer Factorization Problems.* Research web page, http://www.aloul.net/benchmarks.html, [accessed May 2015].

[4] Subbarayan, S., Pradhan, D., K.: *NiVER: Non Increasing Variable Elimination Resolution for Preprocessing SAT Instances.* Proceedings of The 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), pp. 276-291, LNCS 3542, Springer 2005, http://www.itu.dk/people/sathi/niver.html , [accessed May 2015].

[5] Bacchus, F., Winter, J.: *Effective Preprocessing with Hyper-Resolution and Equality Reduction.* Proceedings of the Theory and Applications of Satisfiability Testing, 6th International Conference, (SAT 2003), pp. 341-355, LNCS 2919, Springer 2004, http://www.cs.toronto.edu/~fbacchus/ sat.html, [accessed May 2015].

Article (vi)

This article deals with makespan suboptimal CPF solving via reducing it to SAT. A so-called *reduced time expansion graph*, which is a directed graph derived from the input CPF instance, is suggested in the article. The core property of reduced time expansion graph is that a set of vertex disjoint directed paths connecting certain vertices exists if and only if the given CPF has a (makespan suboptimal) solution. The question of existence of vertex disjoint paths in reduced time expansion graph is modeled as satisfiability of a propositional formula and for its answering a SAT solver[1] is used.

Reduced time expansion graph is based on time expansion of the input graph modeling the environment – that is, the environment graph is copied multiple times and consecutive copies in the sequence of copies – called *layers* – are interconnected by directed edges (copies of the same vertex are connected by directed edges). The trajectory of an agent unambiguously corresponds to a path starting in the initial position within the first layer and terminates in the goal vertex within the final layer. The path may visit multiple vertices within the single layer.

The advantage of reduced time expansion graph is that only few layers are needed if the interaction among agents is limited and hence the resulting propositional formula is small. In the extreme case when agents are completely isolated, the existence of vertex disjoint paths is granted even for single layer of the original graph as it is sufficient to interconnect initial positions and goals by vertex disjoint paths in the graph modeling the environment (the prerequisite is that initial positions and goals are disjoint). Small propositional formulae derived from the CPF instances improve scalability and speed of the solving process.

It has been observed that reduced time expansion graph need few time expansions only (typically one or two) if the difference between the initial and goal arrangement of agents is small. This observation was behind the idea to place agents to their goal positions one by one – that is, a CPF where the next agent is relocated towards its goal is solved using reduced time expansion graph in each step (the process is complete as other agents including those already placed can move freely though they need to return to their positions). Thus, instead of solving one complex CPF instance via SAT a series of very simple instances are solved in which SAT solvers particularly excel. The solving process is called UniAGENT. Interestingly the overall makespan is not compromised even though

[1] Audemard, G., Simon, L.: *The Glucose SAT Solver*, 2013, http://labri.fr/perso/lsimon/glucose/, [accessed May 2015].

agents are placed one by one, since the movement of the next agent is mostly independent on movements of previous agents and can be performed in parallel.

The experimental evaluation indicates that solving CPF makespan sub-optimally via SAT represents a significant relaxation with respect to makespan optimal SAT-based solving[1,2]. The relaxation enables to solve larger instances in terms of the number of agents and size of the graph and represents a good trade-off between quality of makespan of generated solutions and speed of solving – the UniAGENT technique generates solutions of shorter makespan than rule based polynomial time algorithms like BIBOX but it is much faster than SAT-based and other makespan optimal methods.

Article (vii)

This article is also motivated by SAT-based solving of CPF. It turned out that modeling of a so-called ALLDIFFERENT constraint[3] in SAT is especially important when CPF is reduced to a propositional formula. The ALLDIFFERENT constraint requires that each variable from a set of involved finite domain variables takes a different value. An efficient filtering algorithm is supplied with the constraint in *constraint programming* paradigm, which makes it an important modeling construct.

The importance of the ALLDIFFERENT constraint for CPF is that it naturally expresses requirements such as that each agent occupies its own vertex; in other words positions of agents are all different, which can be nicely expressed by the ALLDIFFERENT constraint over variables representing agent's positions.

The assumption is that finite domain variables are translated to bit vectors (vectors of propositional variables) using binary encoding. At the time of writing this article, the commonly used model was to express inequalities between all pairs of bit vectors, which required quadratic number of inequalities. The newly proposed encoding uses completely different approach – it enforces strict linear

[1] Surynek, P.: *Optimal Cooperative Path-Finding with Generalized Goals in Difficult Cases.* Proceedings of the 10th Symposium on Abstraction, Reformulation, and Approximation (SARA 2013), Leavenworth, WA, USA, AAAI Press, 2013.

[2] Standley, T. S., Korf, R. E.: *Complete Algorithms for Cooperative Pathfinding Problems.* Proceedings of Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011), pp. 668-673, IJCAI/AAAI Press, 2011.

[3] Régin, J-C.: *A Filtering Algorithm for Constraints of Difference in CSPs.* Proceedings of the 12th National Conference on Artificial Intelligence (AAAI 1994), pp. 362-367, MIT Press, 1994.

ordering over a permutation of the set of input bit vectors that models pair wise difference as well (strictly linearly ordered numbers are all different). The immediate benefit is that new encoding requires only linear number of inequalities between bit vectors. The experimental evaluation also shown that the novel encoding is more efficient in terms of runtime especially in hard setups of the constraint (it cannot be straightforwardly checked that there is enough or too few values in domains of variables to satisfy the constraint or declare it unsatisfiable).

Article (viii)

A technique that improves algorithms for CPF solving that place agents to their goals one by one is presented in this article. The idea is quite simple; when an agent is relocated towards its goal the next agent to be placed nearby is opportunistically taken on the way and two (or more) agents are relocated together towards their goal – relocated agents form a so-called SNAKE. The effect of the SNAKE-based relocation of agents is that it consumes fewer moves than if agents are relocated separately towards their goals.

In order to show its general applicability, the SNAKE technique has been integrated into the algorithm of Parberry[1] and into the already presented BIBOX algorithm. The prerequisite of using the SNAKE technique is that the target algorithm needs to be open for modification and it places agents in some ordered manner so that newly placed agents are placed near those already placed.

The algorithm of Parberry solves sub-optimally a special case of CPF known as $(N^2 - 1)$-puzzle[2], which is in fact CPF over 4-connected square grid of size $N \times N$ with single unoccupied vertex. The algorithm proceeds in placing agents to their goal positions row by row in the grid and runs in polynomial time (the algorithm is designed for on-line solving of the puzzle), in which it is similar to the BIBOX algorithm.

The SNAKE improvement within the algorithm of Parberry concerns entire relocation of a given agent towards its goal while in the case of BIBOX algorithm the technique is used to relocate agents only towards the entrance of the ear containing the goal vertex and the movement towards the goal inside the ear is done

[1] Parberry, I.: *A real-time algorithm for the $(n^2$-1)-puzzle.* Information Processing Letters, Volume 56 (1), pp. 23-28, Elsevier, 1995.

[2] Korf, R. E., Taylor, L. A.: *Finding Optimal Solutions to the 24-Puzzle.* Proceedings of the 13th National Conference on Artificial Intelligence (AAAI 1996), pp. 1202-1207, AAAI Press, 1996.

in without SNAKE improvement in the standard way. Snakes of length two were used in both algorithms. Using the SNAKE technique with more than two agents turned out to be impractical as the third agent significantly increases the number of possibilities how to form a snake of length three (this observation was also confirmed by an experimental evaluation).

The conducted experimental evaluation indicates that SNAKE-based improvement leads to reduction of the total number of moves in solutions by approximately 8% in case of the algorithm of Parberry and to 30% - 80% reduction in case of the BIBOX algorithm depending on the input instance. Larger improvements of solutions in case of the BIBOX algorithm were achieved in instances with multiple unoccupied vertices and also in graphs whose ear decomposition contains longer ears.

Article (ix)

A great part of author's recent research has been devoted to makespan optimal solving of CPF via reducing it to propositional satisfiability; multiple conference articles regarding this problem were published[1,2,3,4,5]. The presented article represents summary of results achieved and published in the topic so far. Optimality with respect to the makespan is practically important measure as it corresponds

[1] Surynek, P.: *On Propositional Encodings of Cooperative Path-finding.* Proceedings of the 24th International Conference on Tools with Artificial Intelligence (ICTAI 2012), pp. 524-531, Athens, Greece, IEEE Press, 2012.

[2] Surynek, P.: *Mutex Reasoning in Cooperative Path Finding Modeled as Propositional Satisfiability.* Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2013), pp. 4326-4331, Tokyo, Japan, IEEE Press, 2013.

[3] Surynek, P.: *Compact Representations of Cooperative Path-Finding as SAT Based on Matchings in Bipartite Graphs.* Proceedings of the 26th International Conference on Tools with Artificial Intelligence (ICTAI 2014), pp. 875-882, Limassol, Cyprus, IEEE Press, 2014.

[4] Surynek, P.: *A Simple Approach to Solving Cooperative Path-Finding as Propositional Satisfiability Works Well.* Proceedings of the 13th Pacific Rim International Conference on Artificial Intelligence (PRICAI 2014), Gold Coast, Australia, pp. 827-833, Lecture Notes in Computer Science, Volume 8862, Springer, 2014.

[5] Surynek, P.: *Simple Direct Propositional Encoding of Cooperative Path Finding Simplified Yet More.* Proceedings of the 13th Mexican International Conference on Artificial Intelligence (MICAI 2014), Part II, Tuxtla Gutiérrez, Mexico, pp. 410-425, Lecture Notes in Computer Science, Volume 8857, Springer, 2014.

to the total execution time of the solution – this measure is preferred in many scenarios such as *evacuation*, *transportation scheduling*, unit relocation in *military operations* and so on.

The common feature of all the techniques for makespan optimal CPF solving is that they are based on a structure of a so-called *time expansion graph* that is derived from the input graph modeling the environment. Unlike reduced time expansion graph, the time expansion graph is always expanded up to the given time step – that is, there are as many copies of the original environment graph called *layers* as is the given time step bound. Each layer of the expansion graph captures arrangement of agents at the given time step. Edges interconnect consecutive layers and correspond to original edges (that is, if an original edge connects u and v in the environment graph, then there are two directed edges connecting u in the i-th layer with v in the $(i+1)$-th layer and v in the i-th layer with u in the $(i+1)$-th layer); there are no edges between vertices within the single layer.

The core property of the time expansion graph is that a solution of the given makespan exists if and only if there exists a set of *non-overlapping* vertex disjoint paths connecting vertices representing initial positions of agents in the first layer with vertices representing goals in the final layer in the time expansion graph whose number of layers equals to the given makespan. Non-overlapping vertex disjoint paths are those where the set of source vertices at a given layer is disjoint with the set of target vertices at the next layer. It holds that the solution of CPF unambiguously corresponds to the set of non-overlapping vertex disjoint paths.

The search for disjoint paths in the time expansion graph is modeled in SAT – various propositional encodings were suggested; they are called: INVERSE, ALL-DIFFERENT, MATCHING, DIRECT, and SIMPLIFIED encodings. The detailed description of all the encodings is given in the article. Let us note that former three encodings use binary encoding of finite domain variables while the latter two encode variables directly; that is, there is a separate propositional variable for each element in the finite variable's domain. An interesting feature is implemented in the MATCHING encoding where the existence of vertex disjoint paths is checked by a non-overlapping flow in the first stage (the flow corresponds to using anonymous agents). If the flow does not exist then there is no chance that there are non-overlapping vertex disjoint paths with distinguishable agents. The advantage of this approach is that checking possible existence of paths with anonymous flow first is cheaper than with distinguishable agents.

The experimental evaluation indicates that SAT-based makespan optimal CPF solving significantly outperforms alternative A*-based search methods[1] in environments with limited free space and many agents. In such cases, independence detection heuristics used in A*-based search methods cannot detect any independence due to frequent interactions among agents. If only SAT-based methods are compared with each other then encodings that use direct encoding for finite domain variables – that is, DIRECT and SIMPLIFIED encodings – outperform encodings based on binary encoding of finite domain variables.

There are multiple optimal CPF solving methods with respect to the *total cost* of the solution[2,3] (that is, the total number of moves). It is planned to adapt these techniques for generating makespan optimal solutions and compare them with the SAT-based approach.

3. On-going Challenges in CPF and Multi-Robot Planning

There are still many unsolved issues in cooperative path finding and related areas. Except the direct continuation of presented works, there are overlaps of CPF into more general problems with potential great practical applicability.

We recently started research on a so-called *adversarial cooperative path finding*[4,5] (ACPF) which adds an opponent out of our control. Two or more teams of agents compete in reaching given goals in ACPF. The team whose agents reach goals as first is the winner. It is obvious that ACPF adds important adversarial dimension to path finding – the planning system not only need to reach goals by its agents but it also need to thwart effort of the adversary. Operations like block-

[1] Standley, T. S.: *Finding Optimal Solutions to Cooperative Pathfinding Problems.* Proceedings of the 24th Conference on Artificial Intelligence (AAAI 2010), AAAI Press, 2010.

[2] Sharon, G., Stern, R., Felner, A., Sturtevant, N. R.: *Conflict-based search for optimal multi-agent pathfinding.* Artificial Intelligence, Volume 219, pp. 40-66, Elsevier, 2015.

[3] Sharon, G., Stern, R., Felner, A., Sturtevant, N. R.: *Conflict-Based Search For Optimal Multi-Agent Path Finding.* Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI 2012), AAAI Press, 2012.

[4] Ivanová, M., Surynek, P.: *Adversarial Cooperative Path-Finding: A First View.* The 27th AAAI Conference on Artificial Intelligence (AAAI 2013), Bellevue, WA, USA, late breaking track, AAAI Press, 2013.

[5] Ivanová, M., Surynek, P.: *Adversarial Cooperative Path-finding: Complexity and Algorithms.* Proceedings of the 26th International Conference on Tools with Artificial Intelligence (ICTAI 2014), pp. 75-82, Limassol, Cyprus, IEEE Press, 2014.

ing of the adversarial team or occupying its goal vertices need to be taken into account. Hence, tool and concepts of *game theory* seem to be useful to address problems with adversarial behavior in CPF.

This research is obviously very topical as it has great potential in planning of movements of ground military forces and other security operations. At the same, the high theoretical complexity of the problem (the problem has been shown to be *PSPACE*-hard) represents a challenge and encourages to developing practical solving algorithms.

CPF can be regarded as a domain dependent planning problem from the perspective of *automated planning*[1]. Among major features of this domain-dependent problem belongs its great homogeneity – it contains multiple agents, which are all the same with same abilities. This homogeneity enables to bring reasoning about the problem into *abstract mathematical space* of graphs where its combinatorial properties are more accessible. It is an important research question if similar abstract space reasoning can be applied in other domains than is CPF – particularly in the domain of *cooperative multi-robot planning*[2] with multiple robots of the same type. It is expected that multi-robot cooperation motivated by transportation and construction problems[3] represent a great room for future research. Also because of the fact, that contemporary research in multi-robot planning is focused on cooperation between relatively few heterogeneous robots[4].

The first major research issue in cooperative multi-robot planning is finding appropriate reformulations of planning tasks in abstract mathematical spaces to obtain non-trivial simplifications to reveal combinatorial character of these problems. The eventual success in designing efficient multi-robot cooperation involving large number of robots may have significant practical impact. It may bring a new level of productivity in construction and maintenance operations in scales and areas that are currently unreachable or inaccessible by conventional means.

[1] Ghallab, M., Nau, D. S., Traverso, P.: *Automated Planning: theory and practice.* Morgan Kaufmann Publishers, 2004.

[2] LaValle, S. M., Hutchinson, S.: *Optimal motion planning for multiple robots having independent goals.* IEEE Transactions on Robotics and Automation, Volume 14(6), pp. 912-925, IEEE Computer Society, 1998.

[3] Petersen, K., Nagpal, R., Werfel, J.: *TERMES: An Autonomous Robotic System for Three-Dimensional Collective Construction*. Robotics: Science and Systems Conference (RSS), 2011.

[4] Alami, R., Fleury, S., Herrb, M., Ingrand, F., Robert, F.: *Multi-robot cooperation in the MARTHA project.* IEEE Robotics & Automation Magazine, Volume 5(1), pp. 36-47, IEEE Computer Society, 1998.

Impact factor journal articles

Pavel **Surynek**: *Solving Abstract Cooperative Path-Finding in Densely Populated Environments.* Computational Intelligence (COIN), Volume 30, Issue 2, pp. 402-450, Wiley, 2014.

Solving Abstract Cooperative Path-Finding in Densely Populated Environments

Pavel Surynek

Charles University in Prague
Faculty of Mathematics and Physics
Department of Theoretical Computer Science and Mathematical Logic
Malostranské náměstí 25, Praha, 118 00, Czech Republic
pavel.surynek@mff.cuni.cz

Abstract. The problem of cooperative path-finding is addressed in this work. A set of agents moving in a certain environment is given. Each agent needs to reach a given goal location. The task is to find spatial temporal paths for agents such that they eventually reach their goals by following these paths without colliding with each other. An abstraction where the environment is modeled as an undirected graph is adopted – vertices represent locations and edges represent passable regions. Agents are modeled as elements placed in the vertices while at most one agent can be located in a vertex at a time. At least one vertex remains unoccupied to allow agents to move. An agent can move into unoccupied neighboring vertex or into a vertex being currently vacated if a certain additional condition is satisfied. Two novel scalable algorithms for solving cooperative path-finding in bi-connected graphs are presented. Both algorithms target environments that are densely populated by agents. A theoretical and experimental evaluation shows that suggested algorithms represent a viable alternative to search based techniques as well as to techniques exploiting permutation groups on the studied class of the problem.

***Keywords*:** cooperative path-finding, multi-robot path-planning, motion coordination, (N^2-1)-puzzle, N×N-puzzle,15-puzzle, sliding puzzle, domain dependent planning, makespan optimization, *BIBOX*, *BIBOX-θ*.

1. Introduction

A problem of *cooperative path-finding*– CPF (also known from literature as *multi-agent* or *multi-robot path-planning*) [15, 16, 18, 31] is addressed in this work.

The task is to find spatial-temporal paths for movable agents, which can be either mobile robots or some other movable objects, so that they eventually reach given goals without colliding with each other by following these paths. The agents are moving in a certain physical or a virtual environment, which is abstracted as an undirected graph with agents placed in its vertices. Edges of the graph represent passable regions in the environment. The main source of the complexity of the problem arises from the possibility of interactions of agents with the environment and in major part from interactions among agents themselves. The agents need to avoid obstacles in the environment, which is embodied directly in the graph by absence of edges (or vertices), and they must not collide with each other, which is modeled by the constraint that at most one agent is located in a vertex at a time.

CPF is motivated by many real-life tasks ranging from *navigation of a group of mobile robots*, *rearranging of containers in storage* (see Figure 1), or *ship avoidance* to *computer generated imagery* where motion of multiple characters needs to be planned. All these tasks can be modeled as a CPF at a certain level of abstraction. Actually, the top-level abstraction generally adopted in the CPF approach to these tasks uncovers challenges that must be inevitably faced if someone tries to solve these tasks – such as the question if some arrangement of agents can be reached from another one under the given physical constraints.

The *centralized* approach is adopted throughout this work. That is, all the agents and the whole environment are fully observable to the centralized planning mechanism. The individual agents make no decisions by themselves; they merely execute plans found by the centralized planner. This is an approach adopted also in all the relevant related works.

This work is specifically targeted on the case of CPF with environments densely populated by agents. Such a case is challenging from several points of view. As there is limited unoccupied space in the environment, agents cannot move freely towards their goals and are forced to cooperate intensively with each other.

At the same time, it is interesting to study the possibility of parallel movements of multiple agents at once, which may reduce the total execution time of the plan significantly. To study parallelism in CPF a variant of CPF called *parallel CPF* (*pCPF*) is defined. The pCPF variant additionally enables an agent to enter a vertex that which is simultaneously vacated by another agent if certain additional conditions are satisfied. The intended effect of the relaxed requirement on movements is to allow a chain of agents to move at once where only the leading agent needs to enter a currently unoccupied vertex and other agents follow it. Allowing such higher movement parallelism is a more realistic model in certain scenarios – especially in the case where unoccupied space is shrinking towards zero.

1.1. **Related Works**

One of the recent successful approaches to CPF was to search for a spatial-temporal path for each agent separately from other agents. If agents are considered separately, an approach is usually called *decoupled* [18, 19]. These techniques are build around the A* algorithm [14] in most cases which is used to search for a shortest path from the current location of an agent to its goal while spatial temporal paths of other already scheduled agents are considered. The positive aspect of the decoupled approach is that it often finds plans that are near to the optimum with respect to the *makespan* (the total time or the number of steps necessary to execute the plan) as short paths for agents are preferred during the search. On the other hand, these methods are extremely sensitive to prioritizing agents as it can easily happen that the already scheduled agents block paths for not yet scheduled ones. This is one of the major drawback of the WHCA* algorithm [18] which is intrinsically incomplete due to this phenomenon (up to 100 agents in the environment are reported; approximately 10% of the environment is occupied). The incompleteness is getting more prominent on cases with the increasing density of agents (see Section 4).

In [19], authors present a complete and optimal algorithm for CPF, which uses sophisticated heuristics to reduce the search space by detecting that sometimes no cooperation is necessary among agents. The trouble with incompleteness has been thus overcome in this approach. However, this method seems to be targeted on relatively sparsely populated environments where actually agents can travel most of the trajectory towards their goals without interacting with other agents (results for the occupancy of environment less than 10% are reported; up to 60 agents are reported to move in environments containing approximately 800 vertices).

Several techniques for CPF are trying to exploit structural properties of the problem to increase the performance. For instance, graph structures are heavily exploited in [15, 16]. The undirected graph modeling the environment is first decomposed into sub-graphs of some interesting structure such as cliques and others over which various known patterns of rearranging agents can be used. The search for the final plan is then performed over an abstract map whose nodes are represented by the sub-graphs of the original graph (experiments with up to 20 agents in the environments consisting of hundreds of vertices are reported).

Another way to exploit structural properties of the problem is to observe local juxtapositions in the current arrangement of agents. This approach was adopted by authors in [29, 30, 31, 32]. If some important juxtaposition of agents is detected then known rearrangement process is applied to advance the situation towards an arrangement where agents are closer to their goals. These techniques turned out to be successful on environments containing many agents but also providing lot of unoccupied space (hundreds of agents moving in environment consisting of thousands of vertices are reported).

Cooperative path-finding has been also addressed from different perspective than as a task of finding a route from the initial location to the goal. A concept of so-called *direction maps* is introduced in [6, 7] to enable coherent movements of multiple agents in various complex patterns that often arise in computer entertainment (such as agents *patrolling* around some location in an RTS game and so on).

A rich source of related works for CPF is represented by works on *motion planning over graphs* [8, 10, 12, 13, 35]. The term of *pebble motion on graph* (PMG) used in these work denotes the same concept as CPF in fact. Particularly, important results were achieved for a special case of PMG known as (N^2-1)-puzzle or $N \times N$-puzzle [11, 12, 13], which consists of a 4-connected grid of size $N \times N$ with just one vertex unoccupied. Many algebraic and complexity results are known for (N^2-1)-puzzle and for PMG generally (some of them will be discussed and used later). It is for instance known that finding the makespan optimal solution to the (N^2-1)-puzzle is an NP-hard problem [12, 13].

Regarding general PMG, algorithms proving its membership into the P class are given in [8, 35] with asymptotic time complexities and lengths of generated solutions of $O(|V|^3)$ and $O(|V|^5)$ respectively ($G = (V, E)$ is a graph modeling the environment). The former one – which will be denoted as *MIT*[1] algorithm in this work – represents an algebraic approach to CPF exploiting permutation groups. This algorithm is complete and is capable of solving CPF instances irrespectively of the density of population of agents (just one unoccupied vertex is sufficient in the case with *bi-connected graph* [34] to solve all the solvable instances). The algorithm regards the arrangement of agents as a permutation and the desired goal permutation is composed of elementary permutations over triples of agents. The drawback of the MIT algorithm is that it was not designed for tical use and hence generated solutions have typically long makespan from the pragmatic point of view despite the very good theoretical upper bound of $O(|V|^3)$.

1.2. **Contribution, Motivation, and Organization**

The main contribution of this work is a presentation of two scalable makespan sub-optimal algorithms *BIBOX* and *BIBOX-θ* that are designed for solving CPF on bi-connected graphs. From the pragmatic point of view, presented algorithms are primarily targeted on cases with environment densely populated by agents (that is, with limited unoccupied space).

[1] This working name for the algorithm was chosen by us and it was inspired by the fact that the principal author was affiliated with Massachusetts Institute of Technology (MIT) at the time publishing the article [8]. Authors themselves did not use any name for their algorithm.

Although the targeted case of CPF is special, it has a great practical importance since many real-life environments can be abstracted as *2D/3D grids* which are typically bi-connected. Techniques for tackling CPF in highly occupied space are worthwhile in cases when the space is a scarce resource. Consider for example storage where piles of stored items can be automatically reconfigured – Figure 1. Such kind of automation can save lot of space since without such automation the storage must be larger to make all the piles accessible. Occupying large space with buildings have considerable negative environmental and economic impacts (occupied land was in many cases arable and its occupation is difficult to revert if it is possible at all).

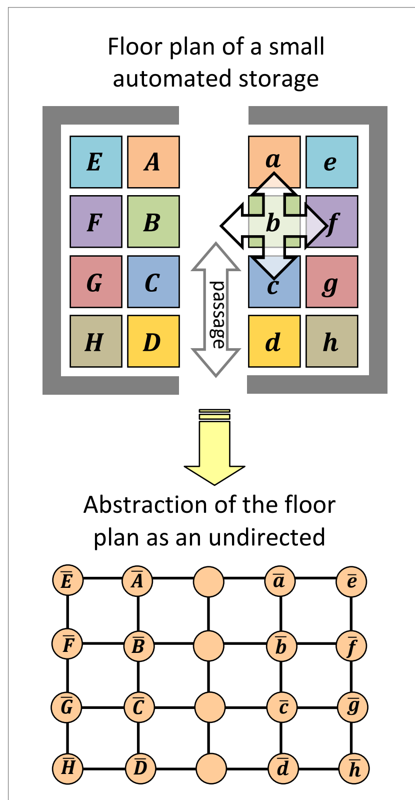


Figure 1. *Illustration of* ***modeling*** *the environment in a real scenario by undirected graph.* The scenario consists of a small automated storage with movable piles of stored items (labeled A to H and a to h). Each pile can be moved left/right/forward/backward. Items in piles are accessible from the passage – to access piles E-H or e-h the storage needs to be rearranged. The environment is modeled as grid of size 4×5 which is a bi-connected graph.

Both suggested algorithms have polynomial time complexity. They were considered as an alternative to the MIT algorithm. A consideration as an alternative to search based algorithms when the makespan optimal solution is not needed and speed of solving is preferred is also viable.

Notice that there is a growing interest in developing algorithms of such category – the very recent contribution represented by the *PUSH-AND-SWAP* algorithm [9] shares lots of aspects with our work (complexity issues and the way of rearranging agents).

Some of the results presented in this work can be also found in some form in conference proceedings [20, 21, 22, 23]. This work is accompanied with a technical report [27] where some additional details such as formal proofs of all the propositions can be found.

The organization of the work is as follows: formal definitions of PMG and pCPF are given first in Section 2. Some basic properties of these problems are discussed subsequently. New algorithms *BIBOX* and *BIBOX-θ* are presented in the main section - Section 3. The final section – Section 4 – is devoted to an extensive experimental evaluation of both new algo-

rithms. A competitive comparison against WHCA* and MIT is presented. Finally, some concluding remarks are given and future prospects are discussed.

2. Pebble Motion on a Graph (PMG) and Parallel Cooperative Path-Finding (pCPF)

Consider an environment in which a group of mobile agents is moving. The agents are all identical (that is, they are all of the same size and have the same moving abilities). Each agent starts at a given initial position and it needs to reach a given goal position. The problem being addressed here consists of finding a spatial-temporal path for each agent so that it eventually reaches its goal by following this path. The agents must not collide with each other and they must avoid obstacles in the environment along the whole process of relocation according to constructed paths.

The environment with obstacles within that the agents are moving is modeled as an undirected graph. The vertices of this graph represent positions in the environment and the edges model passable regions from one position to another. At each time step, all the agents are located in some vertices while at most one agent is allowed per vertex. Some vertices may be vacant – precisely, at least one vertex should be vacant to allow agents to move.

If the agent is placed in a vertex at a given time step then the result of a motion is the situation where the agent is placed in the neighboring vertex at the following time step. The agent is allowed to enter the neighboring vertex supposing it is unoccupied or being vacated by another agent in a certain case while no other agent is trying to enter the same target vertex (precise definition of conditions that the movement must satisfy will follow).

We distinguish two variants of motion problems here, which differ in conditions on movements. Agents in the first one are called *pebbles* and the related problem is called *pebble motion on a graph*. Briefly said, it is required that the target vertex of the movement must be vacant. The second variant is called *parallel cooperative path-finding*. Movable agents in this variant are called *agents* and the condition on movements is relaxed so that it additionally allows movements into vertices that are currently vacated by another agent in a case when agents are moving in a chain style (like a train).

2.1. Formal Definitions of Cooperative Path Planning Problems

The first definition below is for the problem of *pebble motion on a graph*– PMG [8] which is also known as *cooperative path-planning/finding* – CPF [18,19, 29] or *multi-robot/agent path-planning/finding* – MRPP [15, 16, 20, 23]. All these terms from the literature denote the same concept in fact. The special variant of pebble motion on a graph is represented by $(N^2 - 1)$-puzzle (which is also known as the $N \times N$-puzzle) [12, 13].

Definition 1 (***pebble motion on a graph*** **– PMG).** Let $G = (V, E)$ be an undirected graph and let $P = \{\bar{p}_1, \bar{p}_2, \ldots, \bar{p}_\mu\}$ where $\mu < |V|$ be a set of *pebbles*. The *initial arrangement* and the *goal arrangement* of pebbles in G are defined by two uniquely invertible functions $S_P^0: P \longrightarrow V$ (that is $S_P^0(p) \neq S_P^0(q)$ for every $p, q \in P$ with $p \neq q$) and $S_P^+: P \longrightarrow V$ respectively. A problem of *pebble motion on a graph* (*PMG*) is the task to find a number ξ and a sequence of pebble arrangements $\mathcal{S}_P = [S_P^0, S_P^1, \ldots, S_P^\xi]$ such that the following conditions hold (the sequence represents arrangements of pebbles at each time step – the time step is indicated by the upper index):

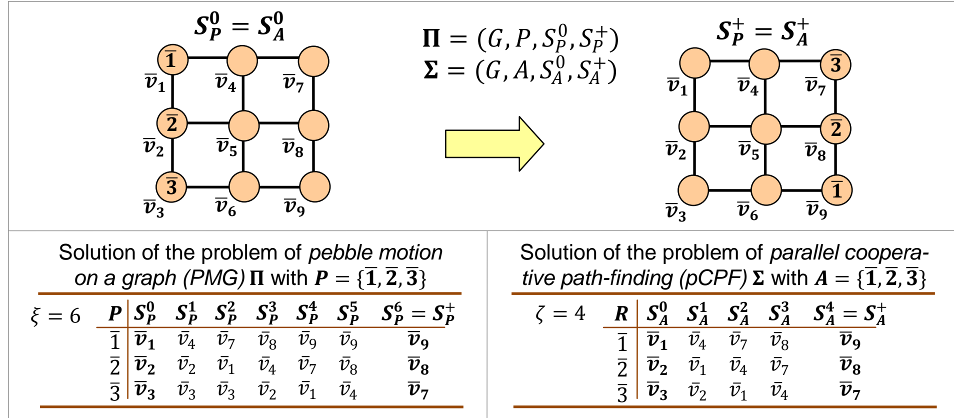
(i) $S_P^k: P \longrightarrow V$ is a uniquely invertible function for every $k = 1,2, \ldots, \xi$;

(ii) $S_P^\xi = S_P^+$ (that is, all the pebbles eventually reach their destination vertices);

(iii) either $S_P^k(p) = S_P^{k+1}(p)$ or $\{S_P^k(p), S_P^{k+1}(p)\} \in E$ for every $p \in P$ and $k = 1,2, \ldots, \xi - 1$ (that is, a pebble either stays in a vertex or moves along an edge);

(iv) if $S_P^k(p) \neq S_P^{k+1}(p)$ (that is, the pebble p moves between time steps k and $k+1$) then $S_P^k(q) \neq S_P^{k+1}(p)$ $\forall q \in P$ with $q \neq p$ must hold for every $p \in P$ and $k = 1,2, \ldots, \xi - 1$ (that is, a pebble can move into a currently unoccupied vertex only).

The instance of PMG is formally a quadruple $\Pi = (G, P, S_P^0, S_P^+)$. A solution to the instance Π will be denoted as $\mathcal{S}_P(\Pi) = [S_P^0, S_P^1, \ldots, S_P^\xi]$. □



Solution of the problem of *pebble motion on a graph (PMG)* $\mathbf{\Pi}$ with $\boldsymbol{P} = \{\bar{1}, \bar{2}, \bar{3}\}$

$\xi = 6$

P	S_P^0	S_P^1	S_P^2	S_P^3	S_P^4	S_P^5	$S_P^6 = S_P^+$
$\bar{1}$	$\bar{v}_1$	$\bar{v}_4$	$\bar{v}_7$	$\bar{v}_8$	$\bar{v}_9$	$\bar{v}_9$	$\bar{v}_9$
$\bar{2}$	$\bar{v}_2$	$\bar{v}_2$	$\bar{v}_1$	$\bar{v}_4$	$\bar{v}_7$	$\bar{v}_8$	$\bar{v}_8$
$\bar{3}$	$\bar{v}_3$	$\bar{v}_3$	$\bar{v}_3$	$\bar{v}_2$	$\bar{v}_1$	$\bar{v}_4$	$\bar{v}_7$

Solution of the problem of *parallel cooperative path-finding (pCPF)* $\mathbf{\Sigma}$ with $\boldsymbol{A} = \{\bar{1}, \bar{2}, \bar{3}\}$

$\zeta = 4$

R	S_A^0	S_A^1	S_A^2	S_A^3	$S_A^4 = S_A^+$
$\bar{1}$	$\bar{v}_1$	$\bar{v}_4$	$\bar{v}_7$	$\bar{v}_8$	$\bar{v}_9$
$\bar{2}$	$\bar{v}_2$	$\bar{v}_1$	$\bar{v}_4$	$\bar{v}_7$	$\bar{v}_8$
$\bar{3}$	$\bar{v}_3$	$\bar{v}_2$	$\bar{v}_1$	$\bar{v}_4$	$\bar{v}_7$

Figure 2. *Example of instance of* ***PMG*** *and* ***pCPF***. Both instances are illustrated on the same graph with the same initial and goal arrangements. The task is to move pebbles/agents from their initial positions specified by S_P^0/S_A^0 to the goal positions specified by S_P^+/S_A^+. A solution of the makespan 6 ($\xi = 6$) is shown for the PMG instance and a solution of the makespan 4 ($\zeta = 4$) is shown for the pCPF instance. Notice the differences in parallelism between both solutions.

When speaking about a move at a time step k, it is referred to the time step of commencing the move (the move is performed instantaneously between time steps k and $k+1$).

The second variant of motion problem on a graph adopted in this work relaxes the condition that the target vertex of a pebble/agent must be vacated in the previous time step. Thus, the motion of an agent entering the target vertex, that is simultaneously vacated by another agent and no other agent is trying to enter the same target vertex, is allowed in a certain case. However, there must be some leading agent initiating such a chain of moves by moving into a currently unoccupied vertex which no other agent is entering at the same time step (that is, agents can move "like a chain" with the leading agent moving into an unoccupied vertex in the front). The problem is formalized in the following definition – it is called *parallel cooperative path-finding – pCPF* since the different style of moving basically enables higher parallelism. The same concept is sometimes also referred as *multi-robot path-planning* in the literature [22, 24, 26, 27].

Definition 2 ***(parallel cooperative path-finding – pCPF).*** Again, let $G=(V,E)$ be an undirected graph. A set of *agents* $A=\{\bar{a}_1,\bar{a}_2,\ldots,\bar{a}_\nu\}$ where $\nu<|V|$ is given instead of the set of pebbles. Similarly, the graph models the environment where the agents are moving. The *initial arrangement* and *the goal arrangement* of agents are defined by two uniquely invertible functions $S_A^0: A \longrightarrow V$ (that is $S_A^0(a) \neq S_A^0(b)$ for every $a,b \in A$ with $a \neq b$) and $S_A^+: A \longrightarrow V$ respectively. A problem of *parallel cooperative path-finding* (*pCPF*) is then the task to find a number ζ and a sequence of agent arrangements $\mathcal{S}_A=[S_A^0,S_A^1,\ldots,S_A^\zeta]$ for that the following conditions hold:

(i) $S_A^k: A \longrightarrow V$ is a valid arrangement for every $k=1,2,\ldots,\zeta$ (that is, uniquely invertible);

(ii) $S_A^\zeta = S_A^+$ (that is, all the agents eventually reach their destinations);

(iii) either $S_A^k(a)=S_A^{k+1}(a)$ or $\{S_A^k(a),S_A^{k+1}(a)\} \in E$ for every $a \in A$ and $k=1,2,\ldots,\zeta-1$ (that is, an agent either stays in a vertex or moves into the neighboring vertex);

(iv) if $S_A^k(a) \neq S_A^{k+1}(a)$ (that is, the agent a moves between time steps k and $k+1$) then there must exist a sequence of distinct agents $[a=b_0,b_1,\ldots,b_\lambda]$ with $\lambda \in \mathbb{N}_0$ such that $S_A^k(c) \neq S_A^{k+1}(b_\lambda)$ $\forall c \in A$ with $c \neq b_\lambda$ (b_λ moves to a vertex that is unoccupied at time step k; b_λ is a *leading* agent of the chain of agents which the sequence is part of) and $S_A^{k+1}(b_i)=S_A^k(b_{i+1})$ for $i=0,1,\ldots,\lambda-1$ (agents $a=b_0,b_1,\ldots,b_{\lambda-1}$ follows the leader like a chain; they move all at once between time steps k and $k+1$).

The instance of pCPF is formally a quadruple $\Sigma=(G,A,S_A^0,S_A^+)$. A solution to the instance Σ will be denoted as $\mathcal{S}_A(\Sigma)=[S_A^0,S_A^1,\ldots,S_A^\zeta]$. □

Notice in point (iv) that if the agent a moves into an unoccupied vertex then the required sequence of distinct agents consists of a itself ($\lambda = 0$) and the latter condition in point (iv) is empty. Notice also that the condition on unique invertibility implies that no two agents can simultaneously enter the same target vertex.

The numbers ξ and ζ represent the *makespan* of solutions. The makespan needs to be distinguished from the *size* of solution, which is the total number of moves performed by pebbles/agents. Example instances of both problems and their solutions are illustrated in Figure 2.

2.2. **Known Properties of Motion Problems and Related Questions**

Notice that a solution of PMG as well as a solution of pCPF allows a pebble/agent to stay in a vertex for more than a single time step. It is also possible that a pebble/agent visits the same vertex several times within the solution. Hence, the sequence of moves for a single pebble/agent does not necessarily form a simple path in the given graph.

Notice further that both problems intrinsically allow parallel movements of pebbles/agents. That is, more than one pebble/agent can perform a move in a single time step. However, pCPF allows higher motion parallelism due to its weaker requirements on agent movements (see Figure 2). More than one unoccupied vertex is necessary to obtain parallelism in PMG while only one unoccupied vertex is sufficient to obtain parallelism within a solution of pCPF (consider for example agents moving around a cycle). The following straightforward proposition puts into relation solutions of instances of PMG and pCPF with the same set of agents and their arrangements over the same graph.

Proposition 1 ***(problem correspondence).*** Let $\Pi = (G, P, S_P^0, S_P^+)$ be an instance of PMG and let $\mathcal{S}_P(\Pi) = [S_P^0, S_P^1, \ldots, S_P^\xi]$ be its solution. Then $\mathcal{S}_A(\Sigma) = \mathcal{S}_P(\Pi)$ is a solution to an instance of pCPF $\Sigma = (G, P, S_P^0, S_P^+)$. ∎

To prove the proposition it is sufficient to observe that the condition (iv) in the definition of pCPF is a relaxation of the corresponding condition in the definition of PMG.

There is a variety of modifications of the defined problems. A natural additional requirement is to produce solutions with the makespan as short as possible (that is, the numbers ξ or ζ are required to be as small as possible). Unfortunately, this requirement makes both PMG and pCPF intractable. It was shown in [12, 13] that the optimization variant of a special case of PMG is NP-hard [3] – this special case is generally known as $N \times N$-*puzzle* or $(N-1)$-*puzzle*. It consists of a graph that can be embedded in the plane as a square 4-connected grid with a single unoccupied vertex. Thus, the optimization variant of general PMG is NP-hard as well.

Here we work with restrictions of both types of problems on *bi-connected graphs* [34]. Hence, it is a reasonable question what is the complexity of these classes. Since the grid graph forming the mentioned $N \times N$-puzzle is bi-connected**,** the immediate answer is that the optimization variant of PMG with a bi-connected graph is NP-hard as well.

Nevertheless, it is not possible to make any similar simple statement about the complexity of the optimization variant of pCPF. The situation here is complicated by the inherent parallelism, which can affect the makespan in some unforeseen way. Constructions used for the $N \times N$ puzzle in [12, 13] thus no longer work. Using different technique it has been recently shown by the author that the optimization variant of pCPF is NP-hard too [24, 26].

Observe further that reported NP-hard case of PMG have a single unoccupied vertex. This fact may raise the question how the situation is changed when there are more than one unoccupied vertices as they may simplify the situation. Unfortunately, it is not the case. PMG with the fixed number of unoccupied vertices is still NP-hard since multiple copies of the $N \times N$ puzzle from [12, 13] can be used to add as many unoccupied vertices as needed. Without providing further details, the instance of pCPF used in the reduction to prove the NP-hardness of the problem in [24] had many unoccupied vertices and its graph was connected (even bi-connected). Altogether, a mere allowance of many unoccupied vertices with no additional structural conditions does not simplify the problem.

Without the requirement on the optimality of the makespan, the situation is much easier; PMG is in the P class as it was shown in [8, 35]. Due to Proposition 1, pCPF is in the P class as well. Thus, it seems that PMG and pCPF have been already resolved. However, constructions proving the membership of PMG into the P class used in [8, 35] generate solutions that are too long for practical use [21, 22, 23]. As the makespan of the solution is of great importance in practice, this fact makes these methods unsuitable when some real life motion problem is abstracted as an instance of PMG. Thus, alternative solving methods has been developed [20, 21, 22, 23] and they are revised in this work.

3. Sub-optimal Solving Algorithms

The basic idea of presented sub-optimal algorithms is to exploit structural properties offered by the concept of *bi-connectivity*. It is known that bi-connected graphs can be inductively constructed as a union of a sequence of *rings* or *handles* while at every stage of this construction the intermediate graph is bi-connected [33, 34].

After arranging agents into the last handle we do not need to care about it anymore and consequently the task reduces to a task of the same type but on a smaller bi-connected graph. Fortunately, bi-connected graphs have another interesting property; every two vertices are connected by at least two vertex disjoint paths, which allow quite complex rearranging of agents. For example, an individ-

ual agent can move relatively freely. One path is traversed by the agent and alternative paths are used to keep unoccupied vertex always in front of the agent. In addition, handles of the decomposition evokes the possibility that agents within them can be rotated, which is actually used in proposed algorithms. Notice that all the mentioned styles of movements are friendly to the parallelism as defined in pCPF – for example, agents in a handle can be rotated within a single time step. However, there are many technical difficulties that need to be addressed to make the above ideas workable.

3.1. **BIBOX: A Novel Algorithm for Pebble Motion on a Bi-connected Graph**

The first algorithm presented here called *BIBOX* was originally proposed in [20]. The input instance should consist of a *non-trivial bi-connected graph* (that is, bi-connected graph not isomorphic to a cycle) with exactly two unoccupied vertices. As the algorithm produces solution consisting of single move per time step it does not matter if PMG or pCPF is given on the input – in the following text pCPF will be always considered. A method how to increase parallelism in the resulting solution to take the advantage of the definition of pCPF will be discussed in Section 3.1.4.

The algorithm proceeds inductively according to the known property of bi-connected graphs that they can be built from a cycle by addition of a sequence of *handles*. Adding a handle means either to insert a new edge into the graph or to connect endpoints of a path consisting of new vertices somewhere into the graph. The important property is that currently built graph is bi-connected at every stage of the construction.

The process of building a graph by adding handles can be reverted as well. That is, the graph can be deconstructed until a cycle remains by removing handles from it. If it is somehow possible to arrange agents whose goal positions are in the handle to be removed before it is actually removed, we have a good starting point for a new solving algorithm because after removal of a handle the problem just reduced to the smaller graph. To obtain a new algorithm it remains to show how agents can be arranged into the handle and how to deal with the cycle that remains at the end of the process.

The process of removing of handles is presented here just for intuition. They actually do not need to be removed during the solving process. It is sufficient not to consider and use a handle after all the agents are properly arranged to their goal positions within that.

The intuition for arranging agents in the cycle that eventually remains is to regard their ordering as a permutation. The goal arrangement of agents in the cycle can be also regarded as a permutation. Thus, we need to change ordering of agents to form another permutation. If it is possible to exchange a pair of agents with respect to their current ordering, then every permutation of agents can be ob-

tained. It will be shown how to utilize two unoccupied vertices to enable exchanges of agents in the remaining cycle.

It is possible to build a bi-connected graph in multiple different ways by adding handles. Hence, the algorithm as well as the produced solution is sensitive to the selection and ordering of handles used in the solving process.

3.1.1. *Graph-theoretical Preliminaries*

The BIBOX algorithm is built around the notion of *bi-connectivity* and around graph theoretical properties of bi-connected graphs [33]. Let us recall the notion of bi-connectivity and related properties briefly.

Definition 3 *(connected graph).* An undirected graph $G = (V, E)$ is *connected* if $|V| \geq 2$ and for any two vertices $u, v \in V$ such that $u \neq v$ there is an undirected path connecting u and v. □

Definition 4 (*bi-connected graph, non-trivial).* An undirected graph $G = (V, E)$ is *bi-connected* if $|V| \geq 3$ and the graph $G' = (V', E')$, where $V' = V \setminus \{v\}$ and $E' = \{\{u, w\} | u, w \in V \wedge u \neq v \wedge w \neq v\}$, is connected for every $v \in V$. A bi-connected graph not isomorphic to a cycle will be called *non-trivial* bi-connected graph. □

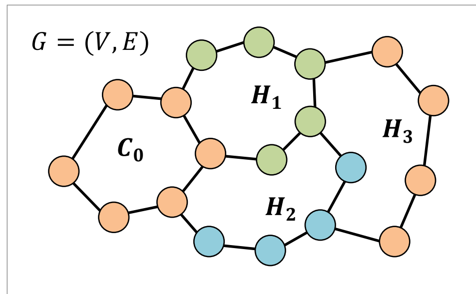


Figure 3. *Example of* ***bi-connected graph****.* A handle decomposition is illustrated.

Observe that, if a graph is bi-connected, then every two distinct vertices are connected by at least two *vertex disjoint paths* (equivalently, there is a cycle containing both vertices; only internal vertices of paths are considered when speaking about vertex disjoint paths - vertex disjoint paths can intersect in their *start points* and *endpoints*). An example of bi-connected graph is shown in Figure 3.

Bi-connected graphs have an important property, which is exploited within the algorithm. Each bi-connected graph can be constructed starting from a cycle by an operation of *adding a handle* [28, 33, 34]. Consider a graph $G = (V, E)$; the new handle with respect to G is a sequence $H = [u, w_1, w_2, \ldots, w_h, v]$ where $h \in \mathbb{N}_0$, $u, v \in V$ (called *connection vertices*) and $w_i \notin V$ for $i = 1, 2, \ldots, h$ (w_i are fresh vertices). The result of the addition of the handle H to the graph G is a new graph $G' = (V', E')$ where $V' = V \cup \{w_1, w_2, \ldots, w_h\}$ and either $E' = E \cup \{\{u, v\}\}$ in the case of $h = 0$ or $E' = E \cup \{\{u, w_1\}, \{w_1, w_2\}, \ldots, \{w_{h-1}, w_h\}, \{w_h, v\}\}$ in the case of $h > 0$. Let the sequence of handles together with the initial cycle be called a

handle decomposition of the given bi-connected graph. Again, see Figure 3 for illustrative example.

Lemma 1 ***(handle decomposition)*** [28, 33, 34]**.** Any bi-connected $G = (V, E)$ graph can be obtained from a cycle by a sequence of operations of adding a handle. Moreover, the corresponding handle decomposition of the graph G can be found in the worst-case time of $O(|V| + |E|)$ and the worst-case space of $O(|V| + |E|)$. ■

The important property of the construction of a bi-connected graph according to its handle decomposition is that the currently constructed graph is bi-connected at every stage of the construction. This property is substantially exploited in the design of the *BIBOX* algorithm.

The algorithm is presented below using a pseudo-code as Algorithm 1 and Algorithm 2 (algorithms are illustrated with pictures for easier understanding). The algorithm starts with the last handle of the handle decomposition and proceeds to the initial cycle. Agents, that goal positions are within the last handle, are moved to their goal positions within this handle. After that, the instance reduces to a smaller bi-connected graph. That is, the last handle is not considered any more since its agents do not need to move any more. This process is repeated until the initial cycle of the decomposition remains where a different technique is used.

Let $\Sigma = (G = (V, E), A, S_A^0, S_A^+)$ be an instance of pCPF. The handle decomposition of the graph G is formally a sequence $\mathcal{D} = [C_0, H_1, H_2, \dots, H_d]$ with $d \in \mathbb{N}$, where C_0 is the initial cycle and H_c is a handle for $c = 1,2, \dots, d$. The order of handle additions in construction of G corresponds to their positions in the sequence (that is, H_1 is added to C_0 first; and H_d is added as the last). A handle $H_c = [u^c, w_1^c, w_2^c, \dots, w_{h_c}^c, v^c]$ for $c \in \{1,2, \dots, d\}$ is assigned a cycle $C(H_c)$. The cycle $C(H_c)$ consists of the sequence vertices on a path connecting v^c and u^c in a graph before the addition of H_c followed by vertices $w_1^c, w_2^c, \dots, w_{h_c}^c$. Specially, it is defined that $C(C_0) = C_0$.

The following lemma justifies two properties exploited by the algorithm. It justifies that it is possible to keep handy two unoccupied vertices in the not yet solved part of the graph since one unoccupied vertex is needed to solve handles and two unoccupied vertices are needed to solve the initial cycle. The lemma ensures that the original goal arrangement can be transformed to an arrangement where unoccupied vertices are located in the initial cycle. Thanks to this property it never happens that an unoccupied vertex become locked in some already solved handle. Details of the transformation are discussed later.

Lemma 2 ***(existence of two vertex disjoint paths).*** Let $G = (V, E)$ be a bi-connected graph and let $u_1, u_2 \in V$ and $v_1, v_2 \in V$, where u_1, u_2, v_1, v_2 are pair-

wise distinct, be two pairs of vertices. Then either the first or the second of the following claims holds:

(a) There exist two vertex disjoint paths φ and χ such that they connect u_1 with v_1 and u_2 with v_2 in G respectively.
(b) There exist two vertex disjoint paths φ and χ such that they connect u_1 with v_2 and u_2 with v_1 in G respectively. ■

Notice that the lemma states that individual vertices in the input pair of vertices are indifferent with respect to connecting by vertex disjoint paths. As the proof of the lemma is rather technical, we refer the reader to [27] where the detailed proof can be found. The idea of proof is that the given 4-tuple of vertices u_1, u_2, v_1, v_2 is assigned a 4-tuple of non-negative integers such that each number refers to a handle of the decomposition or the initial cycle where the corresponding vertex is located. Then the proof proceeds inductively according to the lexicographic ordering of these 4-tuples of numbers. For a selected pair of vertices partial connection paths are constructed towards handles with lower numbers (a certain case analysis in the worst-case time of $O(1)$ has to be done). Then it holds from the induction hypothesis that remaining parts of connection paths should exist since they connect 4-tuple of vertices with lower 4-tuple of assigned numbers.

3.1.2. *Pseudo-code of the BIBOX Algorithm*

Several basic operations are introduced to express the *BIBOX* algorithm in an easier way. These operations are formally described using pseudo-code as Algorithm 1. In addition to functions S_A^0 and S_A^+ there will be a function $S_A\colon A \longrightarrow V$ to represent the current arrangement of agents in G and functions $\Phi_A^0\colon V \longrightarrow A \cup \{\perp\}$, $\Phi_A^+\colon V \longrightarrow A \cup \{\perp\}$, and $\Phi_A\colon V \longrightarrow A \cup \{\perp\}$ which are generalized inverses of S_A^0, S_A^+, and S_A respectively; the symbol $\perp$ is used to represent an unoccupied vertex (that is, $(\forall a \in P)\ \Phi_A(S_A(a)) = a$ and $\Phi_A(v) = \perp$ if $(\forall a \in A)\ S_A(a) \neq v$). Each undirected cycle appearing in the handle decomposition of the input graph is assigned a fixed orientation. Let C be an undirected cycle (a set of vertices of the cycle), then the orientation of C is expressed by functions $next_{\circlearrowright}$ and $prev_{\circlearrowright}$ where $next_{\circlearrowright}(C, v)$ for $v \in C$ is a vertex following v (with respect to the positive orientation) and $prev_{\circlearrowright}(C, v)$ is a vertex preceding v (with respect to the positive orientation). The orientation of a cycle given by $next_{\circlearrowright}$ and $prev_{\circlearrowright}$ is observed also whenever vertices of the cycle are explicitly enumerated in the code.

Each vertex of the input graph is either *locked* or *unlocked*. Auxiliary operations *Lock* (X) and *Unlock*(X) locks or unlocks a set of vertices $X \subseteq V$. The state of a vertex is used to determine whether an agent can move into it. Typically, an agent is not allowed to enter a locked vertex (see the pseudo-code for details).

It is assumed that it holds that $|A| = |V| - 2$ (that is, there are exactly two unoccupied vertices in the graph G). It is required by the main phase of the algorithm that the two unoccupied vertices are located in the first two vertices of the initial cycle within the goal arrangement. This requirement is treated by a function *Transform-Goal* and a procedure *Finish-Solution*. The function *Transform-Goal* determines two vertex disjoint paths from unoccupied vertices in the goal arrangement to the first two vertices in the initial cycle of the handle decomposition. Existence of these paths is ensured by Lemma 2. The goal arrangement is transformed so that finally unoccupied vertices are located in the initial cycle. This is done by shifting agents within the goal arrangement along the two found paths. After the modified instance is solved, the function *Finish-Solution* moves unoccupied vertices back to their goal locations in the original unmodified goal arrangement. The final placement of unoccupied vertices is done by shifting agents along the same two paths but in the opposite direction.

It is assumed that the input bi-connected graph G is non-trivial for further simplifying the pseudo-code; that is, it is not isomorphic to a cycle. The case when the graph is isomorphic to a cycle can be treated easily in a separate branch of the execution.

Several upper level primitives are exploited by the *BIBOX* algorithm. It is possible to make any vertex unoccupied in a connected graph (especially in a bi-connected one) – implemented by procedure *Make-Unoccupied*. Let v be a vertex to be made unoccupied. A path ϕ connecting v and some of the unoccupied vertices avoiding the locked vertices is found. Then agents along the path ϕ are shifted towards the currently unoccupied vertex.

An operation of moving an agent into an unoccupied vertex is implemented by a procedure *Move-Agent-Unoccupied* – the meaning is that the unoccupied space and the agent are swapped. The procedure also updates functions S_A and Φ_A to reflect the new arrangement of agents and constructs the next arrangement S_A^ζ for the output solution sequence.

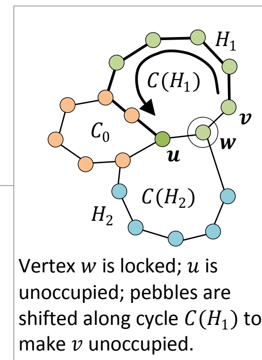
Algorithm 1. *Basic **agent movement** operations.* These operations are used as building blocks for the *BIBOX* algorithm.

---

procedure *Make-Unoccupied*(v)

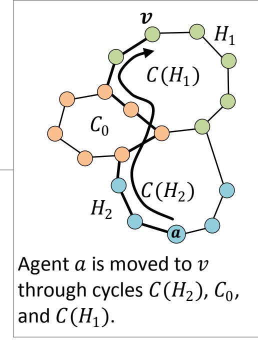
/* Makes a vertex v unoccupied while locked vertices remain untouched.
Parameters: v - a vertex to be made unoccupied. */

1: **let** $u \in V$ such that $\Phi_A(u) = \perp$ and u is not locked
2: **let** $\phi = [u = w_1, w_2, \ldots, w_j = v]$ be a (shortest) path
3: ⌊ connecting u and v in G not containing locked vertices
4: **for** $i = 1,2, \ldots, j-1$ **do**
5: ⌊ Move-Agent-Unoccupied(w_{i+1}, w_i)



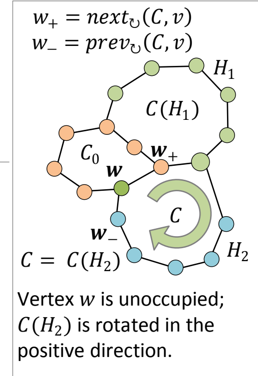
Vertex w is locked; u is unoccupied; pebbles are shifted along cycle $C(H_1)$ to make v unoccupied.

procedure *Move-Agent*(a, v)
/* Moves an agent a into a vertex v
avoiding locked vertices.
Parameters: a – an agent to move,
v - a target vertex.*/
/* complexity issues impose special selection of φ */
1: **let** $\varphi = [S_A(a) = w_1^\varphi, w_2^\varphi, \ldots, w_{j_\varphi}^\varphi = v]$ be a path
2: connecting $S_A(a)$ and v in G not containing
3: locked vertices
4: **for** $i = 1,2, \ldots, j_\varphi - 1$ **do**
5: Lock $(\{w_i^\varphi\})$
6: Make-Unoccupied(w_{i+1}^φ)
7: Unlock$(\{w_i^\varphi\})$
8: Move-Agent-Unoccupied$(w_i^\varphi, w_{i+1}^\varphi)$



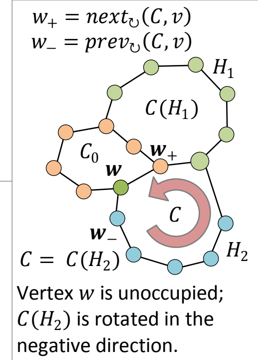
Agent a is moved to v through cycles $C(H_2)$, C_0, and $C(H_1)$.

procedure *Rotate-Cycle*$^+(C, w)$
/* Rotates agents in a cycle C in the positive direction.
Parameters: C - a cycle to rotate
w - unoccupied vertex, $w \in C$. */
1: **for** $i = 1,2, \ldots, |C|$ **do**
2: Move-Agent-Unoccupied$(prev_\circlearrowleft(C, w), w)$
3: $w \leftarrow prev_\circlearrowleft(C, w)$



Vertex w is unoccupied; $C(H_2)$ is rotated in the positive direction.

procedure *Rotate-Cycle*$^-(C, w)$
/* Rotates agents in the cycle C in the negative direction.
Parameters: C - a cycle to rotate, $w \in C$. */
1: **let** $w \in C$ such that $\Phi_A(w) = \bot$ and w is not locked
2: **for** $i = 1,2, \ldots, |C|$ **do**
3: Move-Agent-Unoccupied$(next_\circlearrowleft(C, w), w)$
4: $w \leftarrow next_\circlearrowleft(C, w)$



Vertex w is unoccupied; $C(H_2)$ is rotated in the negative direction.

procedure *Move-Agent-Unoccupied*(u, v)
/* Swaps agent and the unoccupied space; vertex v is supposed to be unoccupied; u contains an agent.
Parameters: u, v – vertices between which agent is moved. */
1: $S_P(\Phi_P(u)) \leftarrow v$
2: $\Phi_P(v) \leftarrow \Phi_P(u)$
3: $\Phi_P(u) \leftarrow \bot$
4: $S_P^\xi \leftarrow S_P$
5: $\xi \leftarrow \xi + 1$

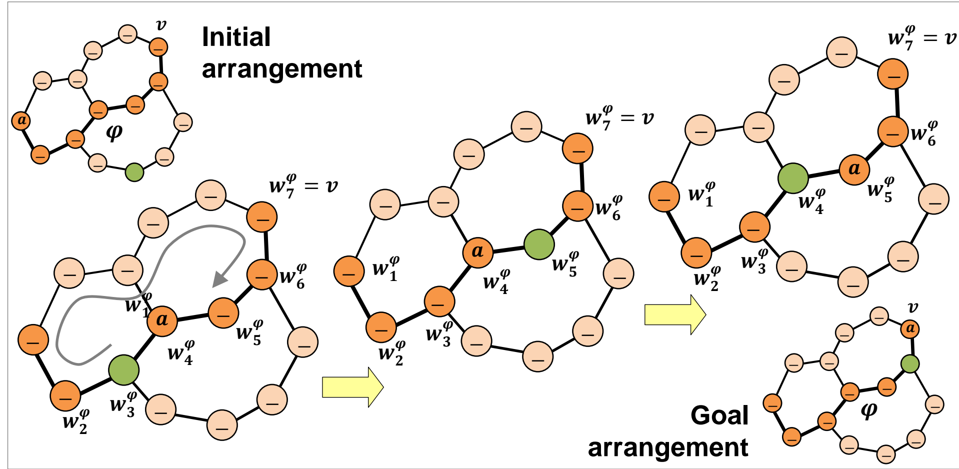


Figure 4. *An illustration of* ***moving*** *an agent in bi-connected graph.* The task is to move an agent a from the initial position to a vertex v. A paths φ connecting the initial position of the agent a with v is found (the path is distinguished by color). It is then traversed by the agent a while the unoccupied vertex is restored in front of a after every edge traversal. This is possible thanks to bi-connectivity of the graph – a path connecting unoccupied vertex and the target vertex avoiding the vertex containing a must always exist. The symbol _ stands for an anonymous agent.

The next important process is moving an agent into a given target vertex. It is implemented by a procedure *Move-Agent*. Let an agent a be moved to a vertex v. A path φ is found such that it connects vertices $S_A(a)$ (which is a vertex currently occupied by a) and v.

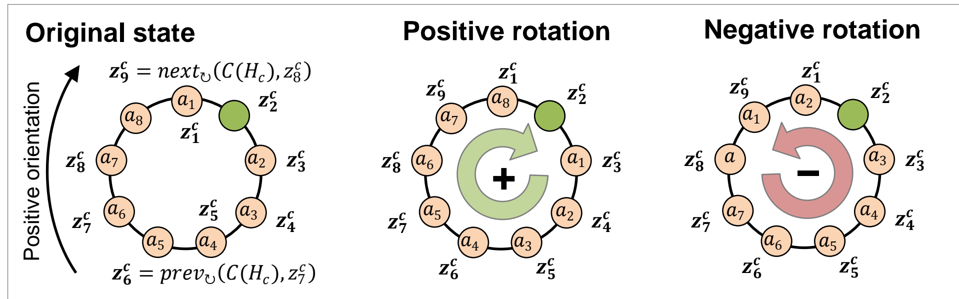


Figure 5. *An illustration of* ***rotation*** *of agents along a cycle.* An orientation of the cycle is determined by functions $next_\circlearrowleft$ and $prev_\circlearrowleft$. There is a single unoccupied vertex in the cycle to enable the rotation.

Edges of φ are then traversed by an agent a. A vertex on φ just in front of a with respect to the direction of the movement is made unoccupied every time a needs to traverse an edge of φ. The agent a should not move during relocation of the unoccupied vertex therefore it is locked before the relocation of the unoccupied vertex starts. Thus, a path along that the unoccupied vertex is moved must

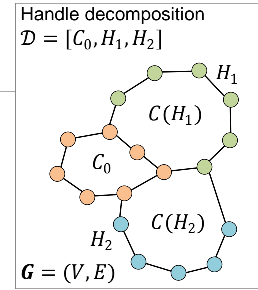
avoid the vertex containing a. Such a path always exists due to the bi-connectivity of the graph in which the relocation of the agent a takes place (see Figure 4 for illustration).

The last basic operation is a rotation of agents along a cycle (see Figure 5). This operation is implemented by procedures *Rotate-Cycle*$^+$ and *Rotate-Cycle*$^-$. The former rotates agents in the positive direction and the latter rotates agents in the negative direction. It supposed that at least one vertex in the given input cycle is unoccupied and it is given as the parameter. The input unoccupied vertex enables the rotation; it remains on its place after the rotation is finished.

Algorithm 2. *The **BIBOX** algorithm.* The pseudo-code is built around operations from Algorithm 1. It solves a given agent motion problem on a non-trivial bi-connected graph with exactly two unoccupied vertices. The algorithm proceeds inductively according to the handle decomposition of the graph of the input instance. The two unoccupied vertices are necessary for arranging agents within the initial cycle of the handle decomposition.

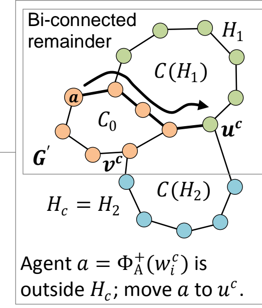
---

function *BIBOX-Solve*$(G = (V, E), P, S_A^0, S_A^+)$: **pair**
/* Top level function of the BIBOX algorithm; solves
a given problem of agent motion on a graph.
Parameters: G - a graph modeling the environment,
A- a set of agents,
S_A^0 - a initial arrangement of agents,
S_A^+- a goal arrangement of agents. */
1: **let** $\mathcal{D} = [C_0, H_1, H_2, \ldots, H_d]$ be a handle decomposition of G
2: $(S_A^+, \varphi, \chi) \leftarrow$ Transform-Goal(G, P, S_A^+)
3: $S_A \leftarrow S_A^0$
4: $\xi \leftarrow 1$
5: **for** $c = d, d-1, \ldots, 1$ **do**
6: **if** $|H_c| > 2$ **then**
7: Solve-Regular-Handle(c)
8: Solve-Original-Cycle
9: Finish-Solution(φ, χ)
10: **return**$(\xi, [S_A^0, S_A^1, \ldots, S_A^\xi])$

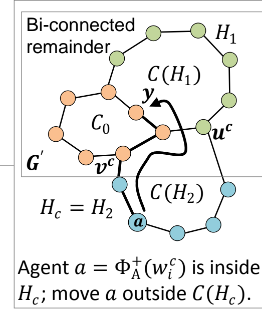


procedure *Solve-Regular-Handle*(c)
/* Places agents which destinations are within a
handle H_c; agents placed in the handle H_c are finally
locked so they cannot move any more.
Parameters: c – the index of a handle */
1: **let** $[u^j, w_1^j, w_2^j, \ldots, w_{h_c}^j, v^j] = H_j \; \forall j \in \{1, 2, \ldots, d\}$
/* Both unoccupied vertices must be located
outside the currently solved handle. */
2: **let** $w, z \in V \setminus \bigcup_{j=c}^{d}(H_j \setminus \{u^j, v^j\})$ such that $w \neq z$
3: Make-Unoccupied(w)
4: Lock$(\{w\})$
5: Make-Unoccupied(z)
6: Unlock$(\{w\})$
7: **for** $i = h_c, h_c - 1, \ldots, 1$ **do**

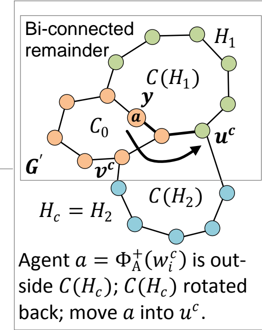
```
8:      Lock(H_c \ {u^c, v^c})
        /* An agent to be placed is outside the handle H_c. */
9:      if S_P(Φ_A^+(w_i^c)) ∉ (H_c \ {u^c, v^c}) then
10:         Move-Agent(Φ_A^+(w_i^c), u^c)
11:         Lock({u^c})
12:         Make-Unoccupied(v^c)
13:         Unlock(H_c)
14:         Rotate-Cycle^+(C(H_c), v^c)
        /* An agent to be placed is inside the handle H_c. */
15:     else
16:         Make-Unoccupied(u^c)
17:         Unlock(H_c)
18:         ρ ← 0
19:         while S_P(Φ_A^+(w_i^c)) ≠ v^c do
20:             Rotate-Cycle^+(C(H_c), u^c)
21:             ρ ← ρ + 1
22:         Lock(H_c \ {u^c, v^c})
23:         let y ∈ V \ (∪_{j=c+1}^{d}(H_j \ {u^j, v^j}) ∪ C(H_c))
24:         Move-Agent(Φ_A^+(w_i^c), y)
25:         Lock ({y})
26:         Make-Unoccupied(u^c)
27:         Unlock(H_c)
28:         while ρ > 0 do
29:             Rotate-Cycle^-(C(H_c), u^c)
30:             ρ ← ρ − 1
31:         Unlock({y})
32:         Lock(H_c \ {u^c, v^c})
33:         Move-Agent(Φ_A^+(w_i^c), u^c)
34:         Lock ({u^c})
35:         Make-Unoccupied(v^c)
36:         Unlock(H_c)
37:         Rotate-Cycle^+(C(H_c), v^c)
38: Lock(H_c \ {u^c, v^c})
```



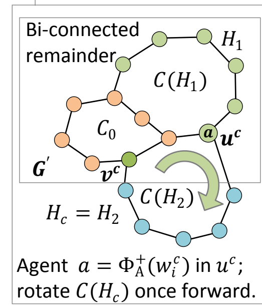
Agent $a = \Phi_A^+(w_i^c)$ is outside H_c; move a to u^c.



Agent $a = \Phi_A^+(w_i^c)$ is inside H_c; move a outside $C(H_c)$.



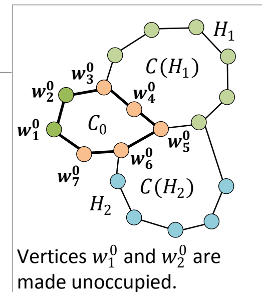
Agent $a = \Phi_A^+(w_i^c)$ is outside $C(H_c)$; $C(H_c)$ rotated back; move a into u^c.



Agent $a = \Phi_A^+(w_i^c)$ in u^c; rotate $C(H_c)$ once forward.

procedure *Solve-Original-Cycle*

```
    /* Places agents which destinations are within the
    initial cycle; it is assumed that unoccupied vertices
    of the goal arrangement of agents are located within
    the initial cycle. */
1:  let u ∈ C_0 and v ∈ V \ C_0 such that {u, v} ∈ E
2:  let [w_1^0, w_2^0, ..., w_l^0] = C_0
    /* According to the assumption on the goal arrangement
    it holds that Φ_A^+(w_1^0) =⊥ and Φ_A^+(w_2^0) =⊥. */
3:  for i = 3,4, ..., l do
4:      Make-Unoccupied(w_1^0)
5:      Lock({w_1^0})
6:      Make-Unoccupied(w_2^0)
7:      Unlock({w_1^0})
8:      if Φ_A^+(w_i^0) ≠ Φ_A(w_i^0) then
9:          Exchange-Agents (Φ_A^+(w_i^0), Φ_A(w_i^0), u, v)
10: Make-Unoccupied(w_1^0)
11: Lock({w_1^0})
```

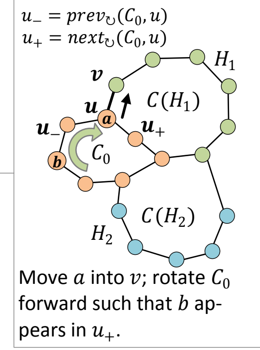


Vertices w_1^0 and w_2^0 are made unoccupied.

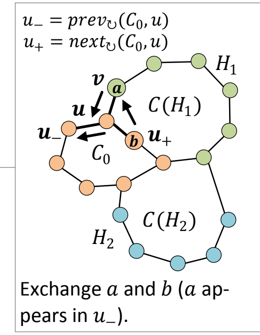
12: Make-Unoccupied(w_2^0)
13: Unlock($\{w_1^0\}$)

procedure *Exchange-Agents*(a, b, u, v)
/* Exchanges a pair of agents within the initial cycle of the handle decomposition.
Parameters: a, b - a pair of agents to be exchanged, u, v - a pair of neighboring vertices where v is used as a storage space. */

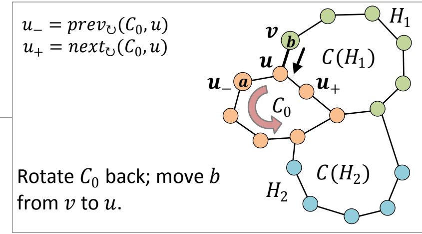
1: $c \leftarrow \Phi_A(v)$
2: Make-Unoccupied(u)
3: Move-Agent-Unoccupied(v, u)
4: **while** $S_A(a) \neq u$ **do**
5: Rotate-Cycle$^+$(C_0, v)
6: Move-Agent-Unoccupied(u, v)
7: Lock($\{u\}$)
8: Make-Unoccupied($next_{\circlearrowleft}(C_0, u)$)
9: $\rho \leftarrow 0$
10: **while** $S_P(b) \neq next_{\circlearrowleft}(C_0, u)$ **do**
11: Rotate-Cycle$^+$(C_0, u)
12: $\rho \leftarrow \rho + 1$
13: Make-Unoccupied($prev_{\circlearrowleft}(C_0, u)$)
14: Move-Agent-Unoccupied(v, u)
15: Move-Agent-Unoccupied($u, prev_{\circlearrowleft}(C_0, u)$)
16: Move-Agent-Unoccupied($next_{\circlearrowleft}(C_0, u), u$)
17: Move-Agent-Unoccupied(u, v)
18: **while** $\rho > 0$ **do**
19: Rotate-Cycle$^-$(C_0, u)
20: $\rho \leftarrow \rho - 1$
21: Move-Agent-Unoccupied(v, u)
22: **while** $S_A(c) \neq u$ **do**
23: Rotate-Cycle$^+$(C_0, v)
24: Move-Agent-Unoccupied(u, v)
25: Unlock($\{u\}$)



Move a into v; rotate C_0 forward such that b appears in u_+.



Exchange a and b (a appears in u_-).



Rotate C_0 back; move b from v to u.

The process of placing agents according to the given goal arrangement is formally described as Algorithm 2. Agents, which goal positions are within the currently solved handle, are placed in a stack like manner. This process is carried out by a procedure *Solve-Regular-Handle* (iteration through the handle is at lines 7-37). Let $H_c = [u^c, w_1^c, w_2^c, \ldots, w_{h_c}^c, v^c]$ for $c \in \{1,2,\ldots,d\}$ be a current handle. Suppose that an agent which goal position is in w_i^c for $i \in \{1,2,\ldots,h_c\}$, that is an agent $\Phi_A^+(w_i^c)$, is processed in the current iteration. Inductively suppose that agents $\Phi_A^+(w_{h_c}^c)$,$\Phi_A^+(w_{h_c-1}^c)$, …,$\Phi_A^+(w_{i+1}^c)$ are located in vertices $w_{h_c-i-1}^c$, $w_{h_c-i-2}^c$,…, w_1^c respectively. An analogical situation for the next agent $\Phi_A^+(w_{i+1}^c)$ must be produced at the end of the iteration.

The agent $\Phi_A^+(w_i^c)$ is moved to the vertex u^c and then the cycle $C(H_c)$ is positively rotated once which causes the agent $\Phi_A^+(w_i^c)$ to move to w_1^c and agents

$\Phi_A^+(w_{h_c}^c)$,$\Phi_A^+(w_{h_c-1}^c)$,…,$\Phi_A^+(w_{i+1}^c)$ stacks in the cycle so that they are located in $w_{h_c-i}^c$, $w_{h_c-i-1}^c$, …, w_2^c. We have just described one iteration of stacking agents into the handle H_c. However, the process has some difficulties. At least, two major cases must be distinguished. In both cases, the first step is that internal vertices of the handle H_c are locked (line 8 of *Solve-Regular-Handle*).

If the agent $\Phi_A^+(w_i^c)$ is not located in the internal vertices of the handle H_c (line 9-14 of *Solve-Regular-Handle*) it is just moved to u^c. This is possible since an invariant holds that both unoccupied vertices are located outside the internal vertices of the handle and the graph without the internal vertices of the handle is connected. This holds at the beginning, since both unoccupied vertices are explicitly moved outside the handle H_c (lines 2-6 of *Solve-Regular-Handle*) and it is preserved through all the iterations. Observe that these movements do not affect agents already stacked in the handle. The agent $\Phi_A^+(w_{h_c}^c)$ is fixed in u^c by locking u^c and then an unoccupied vertex is relocated to v^c which makes the rotation of the cycle $C(H_c)$ possible. The positive rotation of $C(H_c)$ then finishes the iteration.

If the agent $\Phi_A^+(w_i^c)$ is already located in some of the internal vertices of the handle H_c (lines 15-37 of *Solve-Regular-Handle*), the above process is reused but it must be preceded by relocating $\Phi_A^+(w_{h_c}^c)$ outside the handle. The vertex u^c is made unoccupied and the cycle $C(H_c)$ is positively rotated until the agent $\Phi_A^+(w_i^c)$ gets outside the internal vertices of H_c; that is, $\Phi_A^+(w_i^c)$ appears in v^c. Notice, that this series of rotations preserves the order of the already stacked agents. To restore the situation however, the cycle must be rotated back the same number of times. A vertex y outside the already finished part of the graph (that is outside $C(H_c)$ and outside H_j for $j > c$) is selected; the agent $\Phi_A^+(w_i^c)$ is moved into y and it is fixed there by locking.

The vertex u^c is made unoccupied again since the preceding process may move some agent into it (this is possible since w alone cannot rule out the existence of a path from an unoccupied vertex to u^c in the bi-connected graph; there is always an alternative path). The cycle is rotated back so that inductively supposed placement of $\Phi_A^+(w_{h_c}^c)$,$\Phi_A^+(w_{h_c-1}^c)$,…,$\Phi_A^+(w_{i+1}^c)$ is restored. The situation is now the same as in the previous case with $\Phi_A^+(w_i^c)$ outside the handle.

After the last iteration within the handle H_c it holds that the agents $\Phi_A^+(w_{h_c}^c)$,$\Phi_A^+(w_{h_c-1}^c)$,…, $\Phi_A^+(w_1^c)$ are located in vertices $w_{h_c}^c$,$w_{h_c-1}^c$,…,w_1^c respectively. Moreover it holds that unoccupied vertices are both outside the internal vertices of H_c. Thus, the solving process can continue with the next handle in the same way while the already solved handles remain unaffected by the subsequent steps. Notice, that only one unoccupied vertex is sufficient for stacking agents into handles. See Figure 6 for detailed illustration.

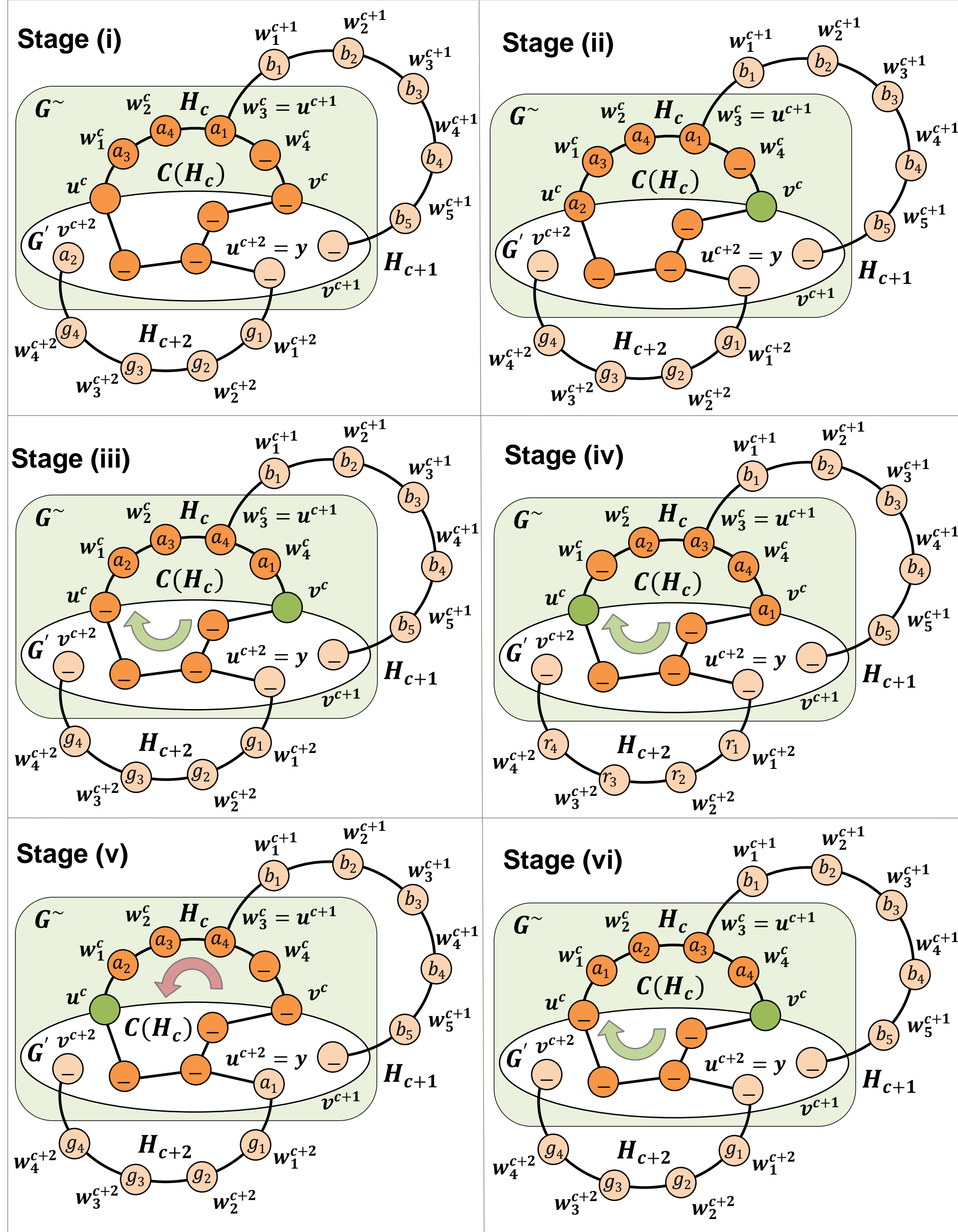


Figure 6. *A process of* ***stacking an agent*** *into a handle.* Agents a_1, a_2, a_3, and a_4 are to be stacked into H_c (that is, $S_A^+(a_1) = w_1^c$, $S_A^+(a_2) = w_2^c$, $S_A^+(a_3) = w_3^c$, and $S_A^+(a_4) = w_4^c$); handles H_{c+1} and H_{c+2} are already solved (that is, $S_A^+(b_1) = w_1^{c+1}$,..., $S_A^+(b_5) = w_5^{c+1}$, and $S_A^+(g_1) = w_1^{c+2}$,..., $S_A^+(g_4) = w_4^{c+1}$). Observe that the agent a_2 is originally outside the handle while the agent a_1 is inside. Stage (i) shows situation after agents a_3 and a_4 were stacked into the handle H_c. Then vacant vertex is relocated to the connection vertex v^c; using empty v^c H_c is rotated such that a_1 appears in the second connection vertex u^c (stages (ii), (iii), and (iv)). The handle H_c then needs to be rotated back but before a_1 must be moved outside the cycle associated with H_c (stage (v)). Finally, the agent a_1 is moved to the first connection vertex u^c and H_c is rotated once so that a_1 appears in the first internal vertex of H_c. The symbol _ stands for an anonymous agent.

The initial cycle C_0 of the handle decomposition must be treated in a different way. Here, the second unoccupied vertex is utilized. An arrangement of agents within C_0 can be regarded as a permutation. The task is to obtain the right permutation corresponding to the goal arrangement. This can be achieved by exchanging several pairs of agents. More precisely, if an agent residing in a vertex of C_0 differs from an agent that should reside in this vertex in the goal arrangement, this pair of agents is exchanged. The process is implemented by a procedure *Solve-Original-Cycle* and by an auxiliary procedure *Exchange-Agents* for exchanging a pair of agents.

The procedure *Exchange-Agents* expects that first two vertices of the initial cycle are unoccupied in the current arrangement. However, the function generally does not preserve this property. Hence, the vacancy of the first two vertices of the initial cycle must be repeatedly restored (lines 4-7 and 10-13 of *Solve-Original-Cycle*). The process of exchanging a pair of agents a and b itself exploits a pair of vertices u and v where these two vertices are connected by an edge and it holds that $u \in C_0 \wedge v \notin C_0$. The vertex v is used as an auxiliary storage place.

The need of two unoccupied vertices is imposed by the fact that an agent from C_0 to be stored in v must be rotated into u first. During this process, some vertex of the cycle must be unoccupied to make the rotation possible and the vertex v must be unoccupied as well to make storing possible.

When exchanging the pair of agents a and b it is necessary to preserve ordering of the other vertices. First, an agent occupying the vertex v is moved into the cycle C_0 in order to make v vacant (lines 1-3 of *Exchange-Agents*). Then the cycle is rotated until the agent a appears in u (since there was an agent in u at the beginning of the rotation, there is always some agent in u after all the rotations) and the agent a is stored in v (lines 4-6 of *Exchange-Agents*). Next, the cycle C_0 is rotated positively so that b appears in $next_{\circlearrowleft}(C_0, u)$ (the next vertex to u with respect to the positive orientation) while the number of rotations is recorded (lines 10-12 of *Exchange-Agents*).

Next, agents a and b are exchanged so that ordering of a in the cycle C_0 is the same as of b before the exchange (lines 13-17 of *Exchange-Agents*). Then, the cycle is rotated in the negative direction recorded number of times so that the place within the cycle where a was originally ordered appears in u; thus b is ordered here (lines 18-20 of *Exchange-Agents*). Finally, the agent that has been located in v before the exchange of agents a and b, is put back into v (lines 22-25 of *Exchange-Agents*).

3.1.3. *Summary of Theoretical Properties and Real-life Extensions*

As the proof of soundness and completeness of the *BIBOX* algorithm are mainly technical, we refer the reader to the appendix where detailed proofs can be found. Regarding the proof of soundness it is necessary to verify that the following step

of the algorithm is always defined particularly at non-deterministic steps where existence of some object – vertex or path – is required (this concerns for example existence of paths at lines 1-3 of *Move-Agent*). Some special care needs to be devoted to verifying that its existence is ensured in the unlocked part of the graph.

It can be shown that the worst-case time complexity of the *BIBOX* algorithm is $\mathcal{O}(|V|^3)$ with respect to the input graph $G = (V, E)$. Again, the detailed proof can be found in the appendix. It needs to be observed that at most $|A|$ agents need to be placed in regular handles. Each agent placement in the handle requires $\mathcal{O}(|V|)$ rotations of the handle and the constant number of relocations of agents (*Move-Agent*). It is not difficult to observe that single rotation by one position requires $\mathcal{O}(|V|)$ steps hence we have $\mathcal{O}(|V|^2)$ steps per handle rotations. For each relocation of an agent, two vertex disjoint paths need to be found which can be done in worst-case time of $\mathcal{O}(|V|)$. Then agent needs to traverse the path. In the worst-case, $\mathcal{O}(|V|)$ edges need to be traversed. An unoccupied vertex needs to be moved in front of the agent per each edge traversal. This has to be done carefully – for example, we cannot afford to search for a path to the front of the agent in the original graph, as it is too much time consuming. Fortunately, the relocation of the unoccupied vertex can be carried out in $\mathcal{O}(|V|)$ steps using the knowledge of the handle decomposition. In total, we have time of $\mathcal{O}(|A||V|^2)$ for placing agents into regular handles.

Regarding the initial cycle, it is needed to observe that at most $|A|^2$ exchanges of pairs of agents are needed while the single exchange consumes $\mathcal{O}(|V|)$ steps. Altogether, the worst-case time complexity is $\mathcal{O}(|V|^3)$. Exactly the same calculation can be done for determining the total number of moves which is also $\mathcal{O}(|V|^3)$. As the total number of moves is the upper bound for the makespan, the makespan of generated solution is $\mathcal{O}(|V|^3)$ as well.

The natural question is how to apply the *BIBOX* algorithm if there are more than two unoccupied vertices in the input instance (that is, $|A| < |V| - 2$). It is easy to adapt the algorithm to utilize additional unoccupied vertices when it is suitable and to ignore them if they are to cause unnecessary movement. The utilizing additional unoccupied vertices is done through replacement of the non-deterministic selection of an unlocked unoccupied vertex (such as that at line 1 of *Make-Unoccupied*) by the selection of the nearest one (this is also done in the real implementation). On the other hand, if for example rotation of a handle is to be done due to unoccupied position in the handle, which is redundant in fact, then such a movement is automatically ignored. More details about this adaptation of the algorithm for sparse environments are given in [27].

Some further optimizations should be used in the real-life implementation to reduce the makespan of the produced solution. Here, various assumption are explicitly enforced in order to make the pseudo-code simpler (for example, the precondition of having first two vertices of the initial cycle of the handle decomposi-

tion unoccupied before a pair of vertices is exchanged within the cycle - lines 4-6 of *Solve-Original-Cycle*). This approach should be avoided and lazier approach should be adopted in the real-life implementation (in the case of exchanging agents, locations of unoccupied vertices should be detected implicitly in subsequent steps by more sophisticated branching of the code). These kind of more complex branching of the algorithm is used in the experimental implementation.

The real-life implementation of procedures *Solve-Regular-Handle* and *Solve-Original-Cycle* should also use more opportunistic selection of vertices to store agents (vertex y - line 23 of *Solve-Regular-Handle* and vertices u, v - line 1 of *Solve-Original-Cycle*). The nearest vertex to the target agent should be always used. Moreover, selection of these vertices within the procedure *Solve-Original-Cycle* should be done not only at the beginning, but also in every iteration of the main loop.

3.1.4. *Making Solution Parallel*

A simple post-processing step needs to be done to obtain parallel solution of pCPF. Suppose to have a solution of Σ – denoted as $\mathcal{S}_A^{\prec}(\Sigma)$ – as a sequence of moves; that is, $\mathcal{S}_A^{\prec}(\Sigma) = [a_1: u_1 \rightarrow v_1; a_2: u_2 \rightarrow v_2; \dots; a_\zeta: u_\zeta \rightarrow v_\zeta]$ with $a_i \in A$ and $u_i, v_i \in V$ meaning that an agent a_i moves from u_i to v_i at time step i. Actually, such a sequential solution is produced by the *BIBOX* algorithm. Now, we need to distinguish which pairs of moves can be executed in parallel and which must be executed one by one sequentially. Following two definitions captures this intuition.

Definition 5 ***(concurrent moves).*** A move $a_k: u_k \rightarrow v_k$; $k \in \{1,2,\dots,\zeta\}$ is *concurrent* with a move $a_h: u_h \rightarrow v_h$; $h \in \{1,2,\dots,\zeta\}$ with $h < k$ if $a_h \neq a_k$, $u_h = v_k \wedge v_h \neq u_k$, and there is no other move $r_\hbar: u_\hbar \rightarrow v_\hbar$ in $\mathcal{S}_A^{\prec}(\Sigma)$ with $h < \hbar < k$ such that $\{u_\hbar, v_\hbar\} \cap \{u_h, v_h, u_k, v_k\} \neq \emptyset$. Concurrent move are denoted as $r_h: u_h \rightarrow v_h \preccurlyeq r_k: u_k \rightarrow v_k$. □

The definition captures the fact that although the moves are interfering they can be executed at the same time. The relation of concurrence is anti-reflexive due to the requirement on different agents involved and anti-symmetric due to the ordering of moves within the sequential solution.

Definition 6 ***(dependent moves).*** A move $a_k: u_k \rightarrow v_k$; $k \in \{1,2,\dots,\zeta\}$ is *dependent* on a move $a_h: u_h \rightarrow v_h$; $h \in \{1,2,\dots,\zeta\}$ with $h < k$ if $\{u_h, v_h\} \cap \{u_k, v_k\} \neq \emptyset$, either $a_h = a_k$ or $u_h \neq v_k \vee v_h = u_k$, and there is no other move $a_\hbar: u_\hbar \rightarrow v_\hbar$ in $\mathcal{S}_A^{\prec}(\Sigma)$ such that $h < \hbar < k$ such that $\{u_\hbar, v_\hbar\} \cap \{u_h, v_h, u_k, v_k\} \neq \emptyset$. The notation of dependence is $a_h: u_h \rightarrow v_h \prec r_k: u_k \rightarrow v_k$. □

The relation of dependence of moves is reflexive and anti-symmetric due to the ordering of moves within the sequential solution. It puts into relation moves that must be executed sequentially as they either concern the same agent or they interfere spatially through shared vertices. Notice that the definition of dependence is complementary to the definition of concurrence.

It is not difficult to show that every function $t: \bigcup \mathcal{S}_R^{\prec}(\Sigma) \longrightarrow \{1,2,\ldots,\zeta\}$ that satisfies conditions that $t(a_h: u_h \rightarrow v_h) < t(a_k: u_k \rightarrow v_k)$ whenever $a_h: u_h \rightarrow v_h \prec a_k: u_k \rightarrow v_k$ and $t(a_h: u_h \rightarrow v_h) \leq t(a_k: u_k \rightarrow v_k)$ whenever $a_h: u_h \rightarrow v_h \preccurlyeq a_k: u_k \rightarrow v_k$ correctly assigns execution time steps to moves with respect to the definition of pCPF. Particular time-step assignment function t can be found by the *critical path* method [14] for instance. Schedule obtained from the critical path method is optimal in certain sense – details are discussed in [27].

3.2. BIBOX-θ: An Algorithm for a Bi-connected Graphs Exploiting Optimal Macros

The drawback of the *BIBOX* algorithm is that it requires at least two unoccupied vertices. Observe that the second unoccupied vertex is necessary only in the last stage where agents are placed into the initial cycle. Thus, if there is only one unoccupied vertex, the *BIBOX* algorithm would be able to place almost all the agents except those whose goal positions are within the initial cycle.

It is possible to apply the *MIT* algorithm [8] to finish placement of agents in the initial cycle. The *MIT* algorithm is capable of solving instances on all the non-trivial bi-connected graphs with just one unoccupied vertex (the instance with just one unoccupied vertex may be unsolvable; indeed, the *MIT* algorithm can detect such a case). Thus if we combine both algorithms, the combined algorithm can proceed as *BIBOX* for placing agents into all the internal vertices of handles and it can proceed as *MIT* over the remaining initial cycle and the first handle. Unfortunately, the process how *MIT* places agents generates excessively long sequences of moves (see experiments in Section 4).

Despite above facts the idea of using alternative solving process for the initial cycle is still promising. Since the initial cycle and the first handle constitute a structurally simple graph (these graphs are called *θ-like graphs* in the following text), it is feasible to try to solve selected instances of pCPF over these graphs makespan optimally. The good candidate instances for optimal solving are those from which an overall solution of any instance over the graph can be composed. Moreover, the optimal solutions to selected instances can be pre-computed and stored in the database for future use. Since solutions from that the overall solution is composed are optimal, it is reasonable to expect that the makespan of the resulting solution will be short as well. Nevertheless, this is a conjecture that should be proven.

3.2.1. *Algebraic Foundation of the Algorithm*

The bi-connected graph, whose handle decomposition consists of an initial cycle and a single handle, represents structurally the simplest bi-connected graphs over that the non-trivial rearrangement of agents is possible supposed there is a single unoccupied vertex (the structurally simpler bi-connected graph is a cycle where only rotations of agents are possible). These graphs will be referred to as *θ-like graphs*.

Definition 7 (*θ-like graph*). Let $X = [\bar{x}_1, \bar{x}_2, \dots, \bar{x}_\alpha]$, $B = [\bar{y}_1, \bar{y}_2, \dots, \bar{y}_\beta]$, and $Z = [\bar{z}_1, \bar{z}_2, \dots, \bar{z}_\gamma]$ be three sequences of vertices satisfying that $\alpha \geq 1 \wedge \beta \geq 2 \wedge \gamma \geq 1$. An undirected graph $\theta(X, Y, Z) = (V_\theta, E_\theta)$ for such three sets is constructed as follows: $V_\theta = X \cup Y \cup Z$ and $E_\theta = \{\{\bar{x}_1, \bar{x}_2\}, \{\bar{x}_2, \bar{x}_3\}, \dots, \{\bar{x}_{\alpha-1}, \bar{x}_\alpha\}; \{\bar{y}_1, \bar{y}_2\}, \{\bar{y}_2, \bar{y}_3\}, \dots, \{\bar{y}_{\beta-1}, \bar{y}_\beta\}; \{\bar{z}_1, \bar{z}_2\}, \{\bar{z}_2, \bar{z}_3\}, \dots, \{\bar{z}_{\gamma-1}, \bar{z}_\gamma\}; \{\bar{x}_1, \bar{y}_1\}, \{\bar{y}_\beta, \bar{z}_\gamma\}, \{\bar{y}_1, \bar{z}_1\}, \{\bar{x}_\alpha, \bar{y}_\beta\}\}$. An undirected graph $G = (V, E)$ is called a *θ-like graph* if there exist three sets of vertices X, Y, and Z as above such that G is isomorphic to $\theta(X, Y, Z)$. □

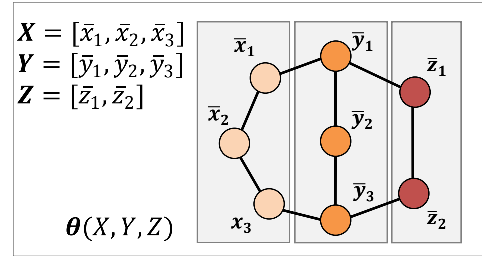


Figure 7. *An example of **θ-like graph**.* θ-like graphs are bi-connected graphs consisting of a cycle and one handle.

The notation of the set union is used over sequences in the definition of the set of vertices V_θ. This is an abbreviation for the union of ranges of individual sequences. Notice that $\theta(X, Y, Z)$ itself is a θ-like graph and $\theta(X, Y, Z)$ may be identical to G if sets X, Y, and Z consist of vertices of G. Hence, no distinction is made between G and $\theta(X, Y, Z)$ in the following text and the notation $\theta(X, Y, Z)$ is used exclusively. An example of θ-like graph is shown in Figure 7.

There are $O(|V|^2)$ non-isomorphic θ-like graphs over a set of vertices V (consider the set V linearly ordered and partitioned over sub-sets X, Y, and Z, where these sub-sets form continuous sub-sequences within the ordered V; there is $O(|V|^2)$ possibilities to place separation points among X, Y, and Z). However, the number of all the possible instances of CPF with a single unoccupied vertex on a fixed θ-like graph $\theta = (V_\theta, E_\theta)$ is $|V_\theta|!$ since the difference between the initial and the goal arrangement can be regarded as a permutation of $|V_\theta|$ elements. Hence, it is not feasible to pre-compute and to store optimal solutions to all the instances of the problem on a fixed θ-like graph. The number of selected instances should be bounded polynomially to make their pre-computation and stor-

ing feasible. At the same time, it should be possible to compose solution to any instance over the θ-like graph from the solutions to selected instances.

Without loss of generality, assume that the unoccupied vertex within the initial and the goal arrangement of an instance over $\theta = (V_\theta, E_\theta)$ is always $\bar{y}_1$ (the unoccupied vertex can be simply relocated to any vertex). The algebraic structure of such instances over θ is isomorphic to the group of all the permutations of $|V_\theta| - 1$ elements which is called a *symmetric group* on $|V_\theta| - 1$ elements and it is denoted $Sym(|V_\theta| - 1)$ [2, 17].

A ***transposition*** is a permutation, which exchanges a pair of elements and keeps other elements fixed. It is well known that $Sym(|V_\theta| - 1)$ can be generated by the set of transpositions. A permutation is called *odd* if it can be composed of an odd number of transpositions. A permutation is called *even* if it can be composed of an even number of transpositions. A permutation is either odd or even but not both. In fact, if a permutation is assigned a *sign* by a sgn function which is $+1$ if the permutation is even and -1 if the permutation is odd, then sgn represents a *group homomorphism* between $Sym(|V_\theta| - 1)$ and the group $(\{-1, +1\}, *, +1, -)$ where multiplication $*$ corresponds to the product of two permutations, neutral element $+1$ corresponds to the identical permutation and unary minus – corresponds to the inverse permutation.

Another simple fact, that can be derived from above statements, is that the set of all an even permutations on the same set of elements forms a proper sub-group of $Sym(|V_\theta| - 1)$; it is called an *alternating group* on $|V_\theta| - 1$ elements and it is denoted as $Alt(|V_\theta| - 1)$.

A *rotation along a 3-cycle* is a permutation that rotates given three elements and keeps other fixed. It is easy to compose any even permutation from rotations along 3-cycles on the same set of elements [8]. As rotation along a 3-cycle itself it is an even permutation it can never generate an odd permutation.

The number of distinct transpositions over n elements is $O(n^2)$ and the number of distinct rotations along 3-cycles over n elements is $O(n^3)$. This is polynomial hence, optimal solutions of corresponding instances seem to be good candidates for storing into the database. Moreover, if the corresponding instances are solvable, then they satisfy the property that a solution to any (in the case of transpositions) or almost any (in the case of 3-cycle rotations) instance on the same graph can be composed of them.

Suppose to have a θ-like graph $\theta(X, Y, Z) = (V_\theta, E_\theta)$ with $X = [\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_\alpha]$, $Y = [\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_\beta]$, and $Z = [\bar{z}_1, \bar{z}_2, \ldots, \bar{z}_\gamma]$ and a set of agents $A = \{a_1, a_2, \ldots, a_{|V_\theta|-1}\}$ for the following three definitions.

Definition 8 ***(even and odd case).*** Let S_A^0 be an initial arrangement of agents such that $S_A^0(a) \neq \bar{y}_1$ $\forall a \in A$ (that is, $\bar{y}_1$ is initially unoccupied) and let S_A^+ be a goal arrangement of agents such that $S_A^+(a) \neq \bar{y}_1$ $\forall a \in A$ (that is, $\bar{y}_1$ is finally unoccu-

pied). If S_A^+ forms an even permutation with respect to S_A^0, then an instance of pCPF $\Sigma = (\theta = (V_\theta, E_\theta), A, S_A^0, S_A^+)$ is called an *even case*. If S_A^+ forms an odd permutation with respect to S_A^0, then the instance Σ is called an *odd case*. □

Definition 9 ***(transposition case).*** Let S_A^0 be an initial arrangement such that $S_A^0(a) \neq \bar{y}_1$ $\forall a \in A$ (that is, $\bar{y}_1$ is initially unoccupied) and let S_A^+ be a goal arrangement such that there exist $b_1, b_2 \in A$ with $b_1 \neq b_2$ for which it holds that $S_A^+(b_1) = S_A^0(b_2) \wedge S_A^+(b_2) = S_A^0(b_1) \wedge (\forall a \in P)(a \neq b_1 \wedge a \neq b_2) \Rightarrow S_A^+(a) = S_A^0(a)$ (agents b_1 and b_2 are to be exchanged while locations of other agents are kept; consequently $\bar{y}_1$ is finally unoccupied). Then an instance of pCPF $\Sigma = (\theta = (V_\theta, E_\theta), A, S_A^0, S_A^+)$ is called a *transposition case* with respect to b_1 and b_2. □

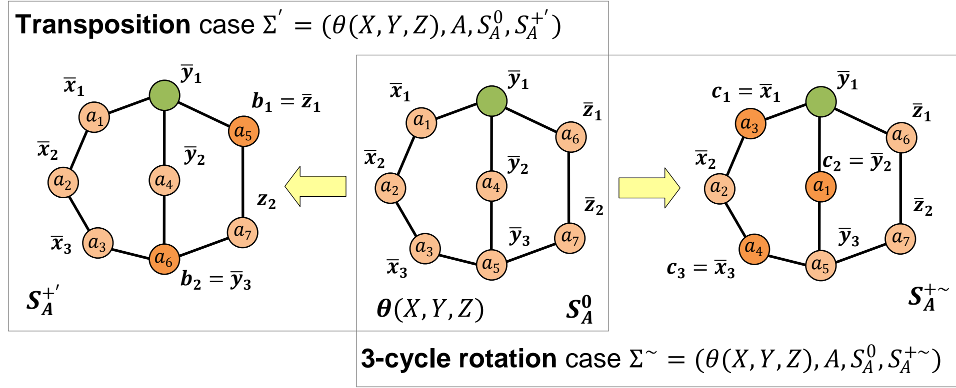


Figure 8. *An example of* ***transposition*** *and* ***3-cycle*** *rotation cases a θ-like graph.* The transposition case is shown for vertices $b_1 = \bar{z}_1$ and $b_2 = \bar{y}_3$. The 3-cycle rotation case is shown for vertices $c_1 = \bar{x}_1$, $c_2 = \bar{y}_2$, and $c_3 = \bar{x}_3$. A solution to any instance over θ-like graph with one vertex unoccupied can be composed of solutions to transposition and 3-cycle rotation cases.

Definition 10 ***(3-cycle rotation case).*** Let S_A^0 be an initial arrangement such that $S_A^0(a) \neq \bar{y}_1$ $\forall a \in A$ ($\bar{y}_1$ is initially unoccupied). Let S_A^+ be a goal arrangement such that there exist pair wise distinct $c_1, c_2, c_3 \in A$ and it holds that $S_A^+(c_1) = S_A^0(c_2) \wedge S_A^+(c_2) = S_A^0(c_3) \wedge S_A^+(c_3) = S_A^0(c_1) \wedge (\forall a \in A)(a \neq c_1 \wedge a \neq c_2 \wedge a \neq c_3) \Rightarrow S_A^+(a) = S_A^0(a)$ (agents c_1, c_2, and c_3 are to be rotated while positions of other agents are kept; $\bar{y}_1$ is finally unoccupied). Then an instance of pCPF $\Sigma = (\theta = (V_\theta, E_\theta), A, S_A^0, S_A^+)$ is called a *3-cycle rotation case* with respect to c_1, c_2, and c_3. □

See Figure 8 for illustrations of transposition case and 3-cycle rotation case. Notice, that both cases would be worthless if they are not solvable. Fortunately, several positive results regarding solvability of these cases are shown in [8]. Following propositions and corollaries recall some of them (without proofs).

Proposition 2 ***(solvability of an odd case)*** [8]**.** An odd case of pCPF $\Sigma = (\theta(X,Y,Z), A, S_A^0, S_A^+)$ with $|X| \neq 2 \vee |Y| \neq 3 \vee |Z| \neq 2$ is solvable if and only if θ contains a cycle of an odd length. ■

Let the θ-like graph $\theta(X,Y,Z)$ with $|X| = 2 \wedge |Y| = 3 \wedge |Z| = 2$ be denoted as $\theta(2,3,2)$. It represents a special case where some instances over it are solvable and some are unsolvable. The case of $\theta(2,3,2)$ will be treated separately.

Since the transposition is an odd permutation, the following corollary is a direct consequence of the above proposition.

Corollary 1 ***(solvability of transposition case)*** [8]**.** A transposition case $\Sigma = (\theta(X,Y,Z), A, S_A^0, S_A^+)$ with $\theta(X,Y,Z)$ non-isomorphic to $\theta(2,3,2)$ is solvable if and only if $\theta(X,Y,Z)$ contains a cycle of an odd length. ■

Proposition 3 ***(solvability of an even case)*** [8]**.** An even case $\Sigma = (\theta(X,Y,Z), A, S_A^0, S_A^+)$ with $\theta(X,Y,Z)$ non-isomorphic to $\theta(2,3,2)$ is always solvable. ■

Analogically, since rotation along 3-cycle is an even permutation, the following corollary is a direct consequence of the above proposition.

Corollary 2 ***(solvability of 3-cycle rotation case)*** [8]**.** A 3-cycle rotation case $\Sigma = (\theta(X,Y,Z), A, S_A^0, S_A^+)$ with $\theta(X,Y,Z)$ non-isomorphic to $\theta(2,3,2)$ is always solvable. ■

Similar results hold not only for θ-like graphs, but also for the more general class of non-trivial **bi-connected** graphs non-isomorphic to $\theta(2,3,2)$ [8]. The important properties directly exploited by the algorithm are that if the input graph does not contain a cycle of an odd length and the initial and the goal arrangement of agents form an odd permutation then the instance is unsolvable. Similarly, if the input and the goal arrangements form an even permutation (and the input graph is non-isomorphic to $\theta(2,3,2)$) then the instance is always solvable (observe that, this is the corollary of the *BIBOX* algorithm and Proposition 3).

The following propositions are important with respect to the length of the overall solution composed of the optimal solutions to the transposition cases and 3-cycle rotation cases. Propositions appeared in [2, 8, 17] but most likely they are just a general knowledge.

Proposition 4 ***(solving an odd case)*****.** A solution to any odd case on a θ-like graph $\theta = (V_\theta, E_\theta)$ can be composed of at most $|V_\theta| - 2$ solutions to transposition cases on the same graph. ■

Similarly, a solution of an even case can be composed of at most $|V_\theta| - 2$ solutions to transposition cases as well.

Proposition 5 ***(solving an even case).*** A solution to any even case on a θ-like graph $\theta = (V_\theta, E_\theta)$ can be composed of at most $|V_\theta| - 2$ solutions to 3-cycle rotation cases on the same graph. ■

Proofs are shown within the pseudo-code of the *BIBOX-θ* algorithm. The above facts justify that transposition and 3-cycle rotation cases are suitable for optimal solving. The corresponding optimal solutions are hence good building blocks for solutions to general instances over θ-like graphs. It is out of scope of this work to give any detailed description of how to compute optimal solutions of instances over θ-like graphs. Applications of several variants of iterative deepening search for this task were studied in [21].

The case of θ-like graph $\theta(2,3,2)$ represents a situation where there is no simple characterization of solvable instances. Since it is a small graph, it is feasible to pre-compute and to store optimal solutions to all the solvable instances over it.

The solving process of the new algorithm over the initial cycle and the first handle is based on the knowledge of how to solve instances over θ-like graphs. In this context, it is necessary to guarantee that insolvability of an sub-instance over $\theta(2,3,2)$ does not contradict solvability of the instance as the whole if the initial cycle and the first handle unluckily become isomorphic to $\theta(2,3,2)$. The following lemma states that this contradictory case can be always avoided. This crucial treatment ensures the upcoming algorithm to proceed correctly. The proof the lemma enumerates all the possible cases and for its length is omitted here (in can be found [27]).

Lemma 3 ***(avoiding*** $\theta(2,3,2)$***).*** If a non-trivial bi-connected graph G is non-isomorphic to $\theta(2,3,2)$ then it subsumes a θ-like sub-graph $\theta(X,Y,Z)$ non-isomorphic to $\theta(2,3,2)$. Moreover, if G contains an odd cycle then it subsumes $\theta(X,Y,Z)$ non-isomorphic to $\theta(2,3,2)$ that additionally satisfies that $2 \nmid |X| + |Y|$ (that is, sets X and Y together form an odd cycle). Having a θ-like sub-graph satisfying above conditions, there exists a handle decomposition of $\mathcal{D} = [C_0, H_1, H_2, \ldots, H_d]$ of G such that $\theta(X,Y,Z) = C_0 \circ H_1$. ($C_0 \circ H_1$ denotes the sub-graph of G constructed by addition of the handle H_1 to the initial cycle C_0). ■

3.2.2. *Pseudo-code of the BIBOX-θ Algorithm*

The new algorithm is called *BIBOX-θ* according to the concept of θ-like graph. Let $\Sigma = (G = (V,E), A, S_A^0, S_A^+)$ be an input pCPF instance on a bi-connected graph with a single unoccupied vertex. If G is non-isomorphic to $\theta(2,3,2)$ and it subsumes a cycle of an odd length then a handle decomposition $\mathcal{D} =$

$[C_0, H_1, H_2, \ldots, H_d]$ of G such that C_0 is of an odd length and $C_0 \circ H_1$ is non-isomorphic to $\theta(2,3,2)$ is computed. Lemma 3 guarantees that this is possible. If G is isomorphic to $\theta(2,3,2)$ then $C_0 \circ H_1$ corresponds to G. If G does not contain an odd cycle then some arbitrary handle decomposition $\mathcal{D}$ is computed.

As in the case of *BIBOX* algorithm, the main phase of the algorithm requires that the finally unoccupied vertex is located in the initial cycle C_0. Thus, a function *Transform-Goal* is applied to modify the goal arrangement S_A^+ by shifting goal locations of agents along a path φ to relocate the unoccupied vertex into C_0. The modified instance is then solved by the process implemented by the *BIBOX-*θ algorithm. The solution is finished by calling a function *Finish-Solution* which shifts agents back along the path φ.

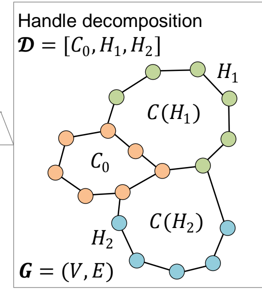
The *BIBOX-*θ algorithm proceeds according to the handle decomposition $\mathcal{D}$ from the last handle H_d to the second handle H_2. The process of placement of agents within the individual handles of the handle decomposition is the same as in the case of the *BIBOX* algorithm. The problem of reaching the goal arrangement of agents within the first handle H_1 and the initial cycle C_0 is solved as an instance over θ-like graph formed by C_0 and H_1. It is supposed that the optimal solutions to all the solvable transposition and 3-cycle rotation cases over θ-like graphs of the size up to the certain limit are pre-computed and stored in the database. Next, it is supposed that the optimal solutions to all the instances over the θ-like graph $\theta(2,3,2)$ are pre-computed into the database as well. A solution to an instance over the θ-like graph is composed of the corresponding optimal solutions stored in the database. If the required record is not stored in the database (which may happen when the size of the θ-like graph is greater than the limit) an alternative solving process must be used. For example, the solving process implemented by the *MIT* algorithm can be employed in such a case.

The pseudo-code of the *BIBOX-*θ algorithm is listed as Algorithm 3. It reuses primitives, functions, and procedures introduced within the context of *BIBOX*. For simplicity, it is supposed that all the required optimal solutions are stored in the database (so there is no treatment when the size of the θ-like graph exceeds the limit).

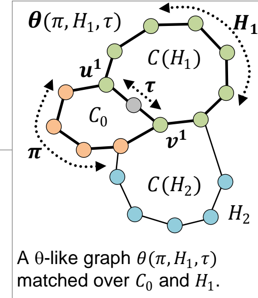
The database with optimal solutions to selected instances over θ-like graphs is represented by three tables: $table_T^\theta$, $table_3^\theta$, and $table_{232}^\theta$. Optimal solutions to transposition cases over a particular θ-like graph θ are stored in the table $table_T^\theta$ – records are addressed by a pair of vertices in which agents are transposed. Similarly, the optimal solutions to 3-cycle rotation cases are stored in the table $table_3^\theta$ – records are addressed by a triple of vertices in which agents are rotated. Finally, the table $table_{232}^\theta$ contains optimal solutions to all the solvable instances over the θ-like graph $\theta(2,3,2)$ - records are addressed by permutations determined by the difference between the initial and the goal arrangement of agents (a function *difference* is used for calculating this differencing permutation).

Algorithm 3. *The **BIBOX-θ** algorithm.* The algorithm solves a given pCPF on a non-trivial bi-connected graph with exactly one unoccupied vertex. It employs a pattern database containing optimal solutions to sub-problems over the initial cycle and the first handle. Functions and procedures from Algorithm 1 and Algorithm 2 are reused here.

function *BIBOX-θ-Solve*$(G = (V, E), A, S_A^0, S_A^+)$: **pair**
/* Top level function of the BIBOX-θ algorithm; solves
a given instance of pCPF with a single unoccupied vertex.
Parameters: G - a graph modeling the environment,
A - a set of agents,
S_A^0 - an initial arrangement of agents,
S_A^+ - a goal arrangement of agents. */
1: **if** G contains a cycle of an odd length **then**
2: **let** $\mathcal{D} = [C_0, H_1, H_2, \dots, H_d]$ be a handle decomposition of G
3: such that C_0 is of an odd length and $C_0 \circ H_1$ is
4: a θ-like sub-graph non-isomorphic to $\theta(2,3,2)$ if possible
/* if this is not possible then G is isomorphic to $\theta(2,3,2)$ */
5: **else**
6: **let** $\mathcal{D} = [C_0, H_1, H_2, \dots, H_d]$ be a handle decomposition of G
/* $C_0 \circ H_1$ is always non-isomorphic to $\theta(2,3,2)$ */
7: $(S_P^+, \varphi) \leftarrow$ Transform-Goal(G, A, S_A^+, C_0)
8: $\xi \leftarrow 1$
9: $S_A \leftarrow S_A^0$

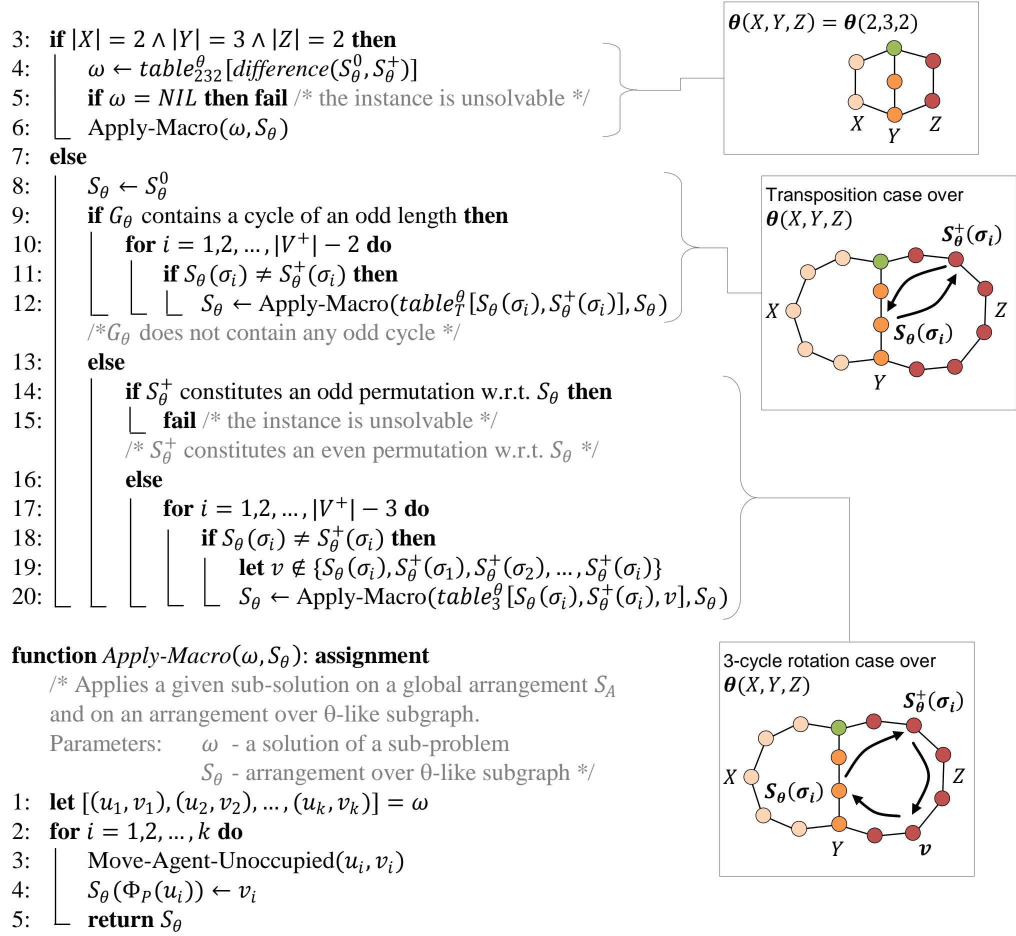


10: **for** $c = d, d-1, \dots, 2$ **do**
11: **if** $|H_c| > 2$ **then**
12: Solve-Regular-Handle(c)
13: **let** $[u^1, w_1^1, w_2^1, \dots, w_{h_1}^1, v^1] = H_1$
14: Lock (V)
15: Unlock$(C_0 \cup H_1)$
16: Make-Unoccupied(u^1)
17: **let** π, τ be two vertex disjoint paths connecting
18: u^1 and v^1 in C_0
19: $\pi \leftarrow \pi \setminus \{u^1, v^1\}$
20: $\tau \leftarrow \tau \setminus \{u^1, v^1\}$
21: θ-BOX-Solve$(\theta(\pi, H_1, \tau), C_0 \cup H_1, S_P, S_P^+)$
22: Finish-Solution(φ)
23: **return** $(\xi, [S_A^0, S_A^1, \dots, S_A^\xi])$



A θ-like graph $\theta(\pi, H_1, \tau)$ matched over C_0 and H_1.

procedure *θ-BOX-Solve*$(\theta(X, Y, Z), V^+, S_\theta^0, S_\theta^+)$
/* Solves a sub-problem over a given θ-like subgraph; a set of
goal vertices into which agents must be placed is specified.
Parameters: $\theta(X, Y, Z)$ - a θ-like subgraph modeling the sub-problem
V^+ - a set of goal vertices
S_θ^0 - an initial arrangement of agents
S_θ^+ - a goal arrangement of agents
(only $S_\theta^+|_{V^+}$ is considered) */
1: **let** $(V_\theta, E_\theta) = \theta(X, Y, Z)$
2: **let** $\{\sigma_1, \sigma_2, \dots, \sigma_{|V^+|-1}\} = \{\sigma | S_\theta^0(\sigma) \in V^+\}$



```
3:  if |X| = 2 ∧ |Y| = 3 ∧ |Z| = 2 then
4:      ω ← table^θ_232[difference(S^0_θ, S^+_θ)]
5:      if ω = NIL then fail /* the instance is unsolvable */
6:      Apply-Macro(ω, S_θ)
7:  else
8:      S_θ ← S^0_θ
9:      if G_θ contains a cycle of an odd length then
10:         for i = 1,2,…,|V^+| − 2 do
11:             if S_θ(σ_i) ≠ S^+_θ(σ_i) then
12:                 S_θ ← Apply-Macro(table^θ_T[S_θ(σ_i), S^+_θ(σ_i)], S_θ)
        /*G_θ does not contain any odd cycle */
13:     else
14:         if S^+_θ constitutes an odd permutation w.r.t. S_θ then
15:             fail /* the instance is unsolvable */
            /* S^+_θ constitutes an even permutation w.r.t. S_θ */
16:         else
17:             for i = 1,2,…,|V^+| − 3 do
18:                 if S_θ(σ_i) ≠ S^+_θ(σ_i) then
19:                     let v ∉ {S_θ(σ_i), S^+_θ(σ_1), S^+_θ(σ_2), …, S^+_θ(σ_i)}
20:                     S_θ ← Apply-Macro(table^θ_3[S_θ(σ_i), S^+_θ(σ_i), v], S_θ)

function Apply-Macro(ω, S_θ): assignment
    /* Applies a given sub-solution on a global arrangement S_A
    and on an arrangement over θ-like subgraph.
    Parameters:    ω  - a solution of a sub-problem
                   S_θ - arrangement over θ-like subgraph */
1:  let [(u_1, v_1), (u_2, v_2), …, (u_k, v_k)] = ω
2:  for i = 1,2,…,k do
3:      Move-Agent-Unoccupied(u_i, v_i)
4:      S_θ(Φ_P(u_i)) ← v_i
5:  return S_θ
```


The main framework of the algorithm as it was described above is represented by the function *BIBOX-θ-Solve* which gets a pCPF instance on a non-trivial bi-connected graph $\Sigma = (G = (V, E), A, S_A^0, S_A^+)$ with just a single unoccupied vertex as a parameter and returns the length of the solution and the solution itself. The difference from the original *BIBOX* algorithm is that the handle decomposition is computed with a special care (lines 1-6) and the final solving process (lines 13-21) over the θ-like graph formed by C_0 and H_1 exploits the solution database. The middle section of the whole solving process (lines 10-12), when agents are placed into handles, is the same as in the case of the *BIBOX* algorithm. To mitigate the need of care about the location of an unoccupied vertex, the first connection vertex of the handle H_1 is vacated (lines 14-16) – this vertex correspond to the vertex $\bar{y}_1$ from the definition of the θ-like graph. Recall, that the transposition, the 3-cycle rotation, and the case of $\theta(2,3,2)$ suppose the unoccupied vertex to be right there.

An auxiliary function *Apply-Macro* is used to apply a record ω from the solution database (the optimal solution for a sub-instance is called a *macro* in this context) on the current arrangement of agents S_θ in a given θ-like graph as well as on the global current arrangement represented by S_A and Φ_A. The optimal solution has the form of a sequence of moves where the move is an ordered pair of vertices of G - the first vertex contains an agent to be moved; the second vertex is unoccupied at the time step of execution of the move and represents the target vertex. The execution of the macro over the current arrangement is carried out by *Move-Agent-Unoccupied*; the function also constructs the next step in construction of the output solution.

The very novel part in comparison with the *BIBOX* algorithm is the process of reaching the goal arrangement over a θ-like graph. This is represented by a function *θ-BOX-Solve*. The function gets as parameters the θ-like graph itself as $\theta(X, Y, Z)$, an initial and a goal arrangement of agents as S_θ^0 and S_θ^+ respectively, and a set of goal vertices as V^+ which is a sub-set of vertices of θ.

If θ is isomorphic to $\theta(2,3,2)$ (lines 3-6) then the goal arrangement is reached at once using a record from the database. It may happen that the required record is not found in the database (line 5). In such a case, the algorithm terminates with the answer that the given instance is unsolvable. A special function *difference* is used in this execution branch. The function calculates a permutation from two arrangements of agents. The interpretation of a permutation calculated by the *difference* function is that it transforms an arrangement given as the first argument to an arrangement given as the second argument.

If θ is non-isomorphic to $\theta(2,3,2)$ and it contains an odd cycle (lines 7-12) then all the goal arrangements are reachable. The goal arrangement is reached by composing several transposition cases. This is done by traversing the set of agents that should be placed. If the current location of an agent given by S_θ differs from its goal location given by S_θ^+, then agents at these two locations are exchanged using a solution for the transposition case from the database of solutions.

If θ is non-isomorphic to $\theta(2,3,2)$ and all the subsumed cycles are of an even length (lines 14-20) then the treatment of unsolvable cases must be done. If the goal arrangement S_θ^+ forms an odd permutation with respect to the initial arrangement S_θ then the given instance is unsolvable (lines 14-15). The algorithm terminates with the negative answer in such a case. If this is not the case (that is, S_θ^+ forms an even permutation with respect to S_θ) then the goal arrangement is reached using 3-cycle rotations (lines 17-20).

This is done almost in the same way as in the case of transposition cases in fact. Again, agents that should be relocated are traversed. The relocation of an agent σ_i to its goal location $S_\theta^+(\sigma_i)$ from $S_\theta(\sigma_i)$ is done by a rotation along a 3-cycle formed by $S_\theta(\sigma_i)$, $S_\theta^+(\sigma_i)$, and v, where v is a vertex different from $S_\theta(\sigma_i)$, $S_\theta^+(\sigma_i)$, and also different from all the goal vertices of all the already placed

agents. Notice, that it is sufficient to traverse all the agents except last two. They must be inevitably placed to their goal vertices after the last 3-cycle rotation since otherwise the goal arrangement S_θ^+ forms an odd permutation with respect to S_θ which has been ruled out at the beginning of this branch.

3.2.3. *Summary of Theoretical Properties and Extensions of the BIBOX-θ Algorithm*

The detailed theoretical analysis of soundness and completeness of the *BIBOX-θ* algorithm can be found in [27]. The crucial ingredient for the correctness of the algorithm is represented by Lemma 3.

The worst-case time complexity of the algorithm is $\mathcal{O}(|V|^4)$ [27] with respect to the input instance $\Sigma = (G = (V, E), A, S_A^0, S_A^+)$. The makespan is also $\mathcal{O}(|V|^4)$ [27]. This result can be obtained from the fact that the length of optimal solutions of special cases is bounded by $(|V|^3)$ [8]. As $\mathcal{O}(|V|)$ optimal solutions of special cases are necessary, the upper bound of $\mathcal{O}(|V|^4)$ is obtained.

If the size of the database containing optimal solutions is not accounted, the space required by the algorithm is of $\mathcal{O}(|V| + |E|)$ in the worst-case. The space required by the part of the database where optimal solutions to $\theta(2,3,2)$ are stored is $\mathcal{O}(1)$ (the size of $table_{232}^\theta$) and the space required by the part of the database where solutions to transposition and 3-cycle rotation cases over a θ-like graph $\theta(X, Y, Z) = (V_\theta, E_\theta)$ are stored is $\mathcal{O}(|V_\theta|^5)$ (the size of $table_T^\theta$) and $\mathcal{O}(|V_\theta|^6)$ (the size of $table_3^\theta$) respectively.

Practically, it is better to use slightly adapted special cases. Observe that special cases as described above preserve all the agents except the affected pair or triple at their original positions. This is not necessary in fact, since only agents that already reached their goal positions need to be preserved. Preserving other agents just imposes additional constraints on the solution and may prolong it unnecessarily. The no less important fact is that it is easier to find a less constrained optimal solution. The modified special cases, where relocation of agents that have not yet reached their goal positions are neglected, are called a *weak transposition case* and a *weak 3-cycle rotation case* respectively. The detailed description of weak special cases is given in [27].

4. Experimental Evaluation

As algorithms *BIBOX* and *BIBOX-θ* were primarily developed as an alternative to the *MIT* algorithm [8], the experimental evaluation will be primarily aimed on the competitive comparison of *BIBOX* and *BIBOX-θ* with *MIT*. Nonetheless, we also provide comparison with the *WHCA** algorithm [18] to obtain more complete image.

All the tested algorithms were implemented in C++. The implementation of algorithms *BIBOX* and *BIBOX-θ* follows the presented pseudo-code. Several optimizations mentioned in Section 3.1.3 were adopted in the implementation of *BIBOX* and *BIBOX-θ* algorithms as well.

The database of optimal solutions used by the *BIBOX-θ* algorithm has been generated on-line (on demand) by a variant of IDA* algorithm enhanced with learning [21]. Details of this algorithm are out of scope of this study. Pseudo-code and experimental analysis can be found in [21]. Notice, that it is a time consuming task to find an optimal solution to a pCPF instance even on a small θ-like graph. Therefore, the timeout of 8.0 seconds was used after that the solving process switched to the *MIT* style. The database with optimal solutions should be pre-computed off-line in the real-life applications.

The *MIT* has been re-implemented according to [8]. The algorithm is designed for general graphs, however the major technique concerns bi-connected partitions. Briefly said, the algorithm finds a configuration of vertices in the input graph on that a 3-cycle rotation is possible. At the same time, it is ensured that every triple of agents can be relocated to this configuration and back to their original locations. By composing these three basic operations – relocation to the 3-cycle rotation configuration, 3-cycle rotation there, and relocation back to original locations – we are actually able to make 3-rotation of every triple of agents. This consequently means that agents can be relocated according to every even permutation by the outlined process (see also Proposition 5). If additionally there is an odd cycle in the input graph, all the permutations are possible.

Similar optimization techniques as in the case of the *BIBOX* algorithm have been used. When an unoccupied vertex was necessary, the nearest unoccupied vertex was found and relocated to the location where needed. More details about the re-implementation of the *MIT* algorithm can be found in [23].

The WHCA* algorithm was also re-implemented by ourselves. It searches for a path for each agent individually while spatial-temporal positions occupied by the already scheduled agents are avoided. This algorithm is inherently incomplete since some agents may block another agent and prevent it from moving; thus, only few of tested setups were solvable by this algorithm.

In order to allow reproducibility of all the presented results the source code and supporting data is provided at the web site: http://ktiml.mff.cuni.cz/~surynek/research/j-multirobot-2010. Additional experimental results and raw experimental data are provided as well.

Experimental evaluation has been performed on two computers. The first computer has been used to generate experimental results regarding runtime - *run-*

time configuration [1]; the second computer has been used to generate all the remaining results - *default configuration*[2].

4.1. Makespan Comparison

The first series of experiments is devoted to comparison of the makespan of solutions generated by tested algorithms. All the tested algorithms were used to generate a sequential solution of a given instance, which has been parallelized subsequently by the critical path method. The result was a parallel solution complying with the definition of the solution of pCPF. A set of testing instances of pCPF consists of instances on *randomly generated bi-connected* graphs and of instances on *grids*.

A randomly generated bi-connected graph has been generated according to its handle decomposition. First, a cycle of random length from *uniform distribution* where certain minimum and maximum lengths were given has been generated. Then a sequence of handles of random lengths from uniform distribution (again the minimum and the maximum length of handles was given) has been added. Each handle has been connected to randomly selected connection vertices in the currently constructed graph. The addition of handles has terminated when the required size of the graph has been reached. An instance on a randomly generated graph itself further consists of random initial arrangement and goal arrangement of agents over the graph where at least the given number of vertices remains unoccupied. The handle decomposition used by solving algorithms was exactly that one used for generating the graph.

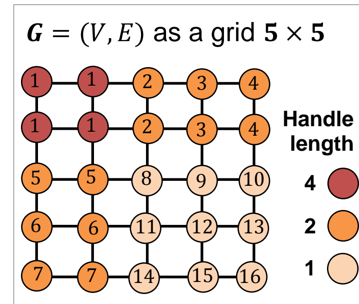


Figure 9. *An illustration of* ***handle decomposition of a grid*** *graph*. The ordering of the addition of individual handles is depicted by numbers in vertices. Three types of handles/cycles are used.

The situation with instances over the grid is similar. The square 4-connected grid graph of a given size has been generated together with a random initial and a goal arrangement of agents. Again, a given number of vertices remain unoccupied. First, an initial cycle with 4 vertices was constructed (placed on the left upper corner of the grid); then handles were added to fill in the grid successively according to its rows and columns. The first row and the first column were added at the beginning (handles with 2 internal vertices). Then rows of the grid were

[1] ***Runtime*** *configuration*: 2x AMD Opteron 1600 MHz, 1GB RAM, Mandriva Linux 10.1, 32-bit edition, gcc version 3.4.3, compilation with –O3 optimization level.

[2] ***Default*** *configuration*: 4x AMD Opteron 1800 MHz, 5GB RAM, Mandriva Linux 2009.1, 64-bit edition, gcc version 4.3.2, compilation with –O3 optimization level.

constructed by adding handles from the left to the right and from the top to the bottom (handles with 1 internal vertex). See Figure 9 for the ordering of addition of vertices in the construction of the grid.

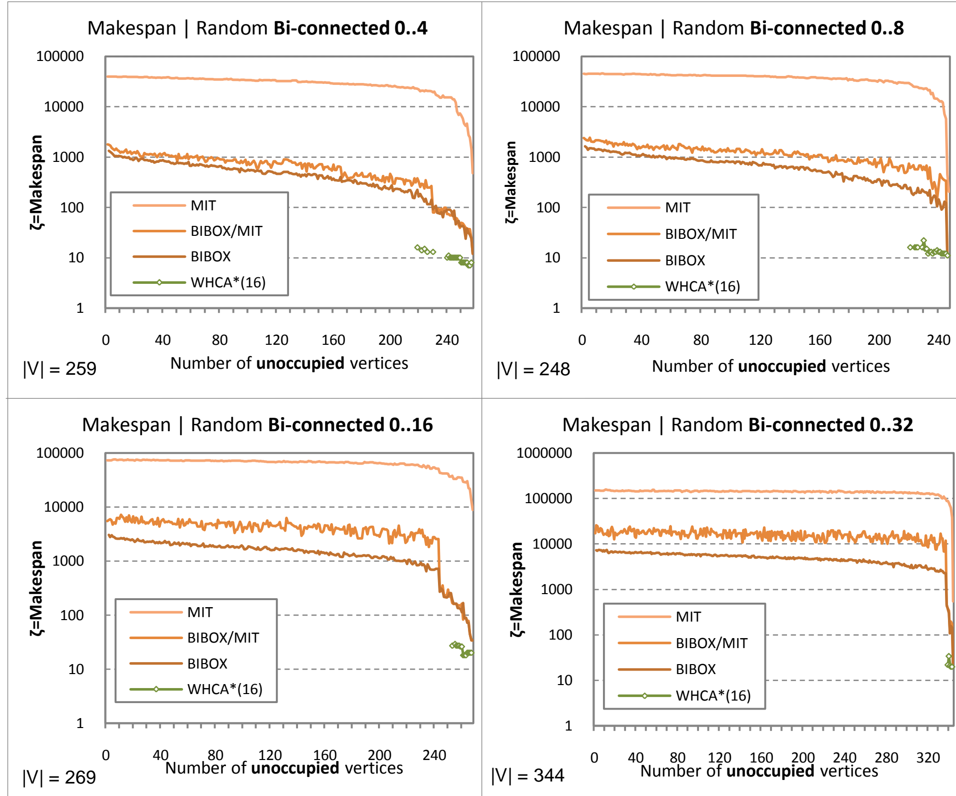


Figure 10. ***Makespan*** *comparison of solutions to instances over* ***random bi-connected*** *graphs.* Four algorithms are compared: the standard *BIBOX*, a variant of *BIBOX* where the last phase when agents are placed into the θ-like graph is solved by *MIT* – *BIBOX/MIT*, the *MIT* algorithm, and WHCA* with the window size of 16. Solutions were parallelized using the presented parallelism-increasing scheme [27] (critical-path method). Four setups of random bi-connected graphs are shown – random lengths handles have uniform distribution of the range: 0..4, 0..8, 0..16, and 0..32 respectively. The makespan tends to decrease for the increasing number of unoccupied vertices. WHCA* was able to solve only several sparsely populated instances.

Results shown in Figure 10 and Figure 11 are targeted on the comparison of the makespan. Results in Figure 10 show makespans of solutions of instances over randomly generated bi-connected graphs. Graphs of size up to 344 vertices were used (the graph had been grown by addition of handles until the size of 256 vertices had been reached). Four graphs, which differ in the average length of the initial cycle and handles of the handle decomposition, were used. Lengths of the initial cycle and handles have the uniform distribution of the range: 0..4, 0..8, 0..16, and 0..32. The length of the handle is equal to the number of its internal

vertices. Figure 11 is devoted to structurally regular graphs – grid graphs of the size 8 × 8, 16 × 16, and 32 × 32 were used.

Four algorithms were compared: the standard *BIBOX*, a variant of *BIBOX* where the last phase when agents are placed into the θ-like graph was solved by *MIT* – *BIBOX/MIT*, the *MIT* algorithm, and *WHCA** with the window size of 16.

Random initial and goal arrangements are obtained as a random permutation of agents in the vertices of the graph. The random permutation is generated from identical one by applying quadratic number of transpositions. This process generates random arrangements of the appropriate quality (of randomness) for the use in the test.

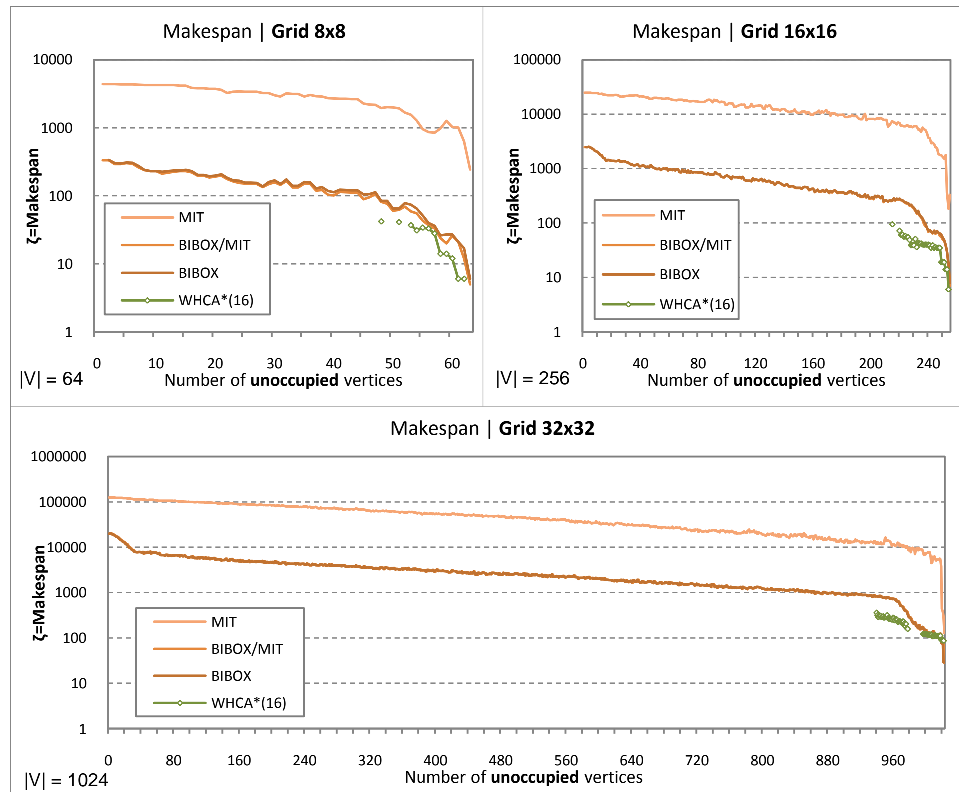


Figure 11. ***Makespan*** *comparison of solutions of instances over* ***square grids***. Four algorithms are compared: the standard *BIBOX*, *BIBOX/MIT, MIT,* and *WHCA**(16) on three grids: 8 × 8, 16 × 16, and 32 × 32.

It can be observed that the *BIBOX* algorithm generates solutions of the makespan approximately 10 times to 100 times smaller than that of solutions generated by the *MIT* algorithm. In the setup with random bi-connected graphs, the difference between *BIBOX* and *MIT* is becoming smaller as the size of handles increases. In the setup with the grid graph, the *BIBOX* algorithm generates solutions that

have approximately 10 times smaller makespan than that of the *MIT* algorithm. A steep decline of the makespan can be observed when the portion of unoccupied vertices reaches approximately 95%. This is some kind of a phase transition when agents are becoming arranged sparsely enough over the graph so that there are almost no interactions between them (that is, they do not need to avoid each other).

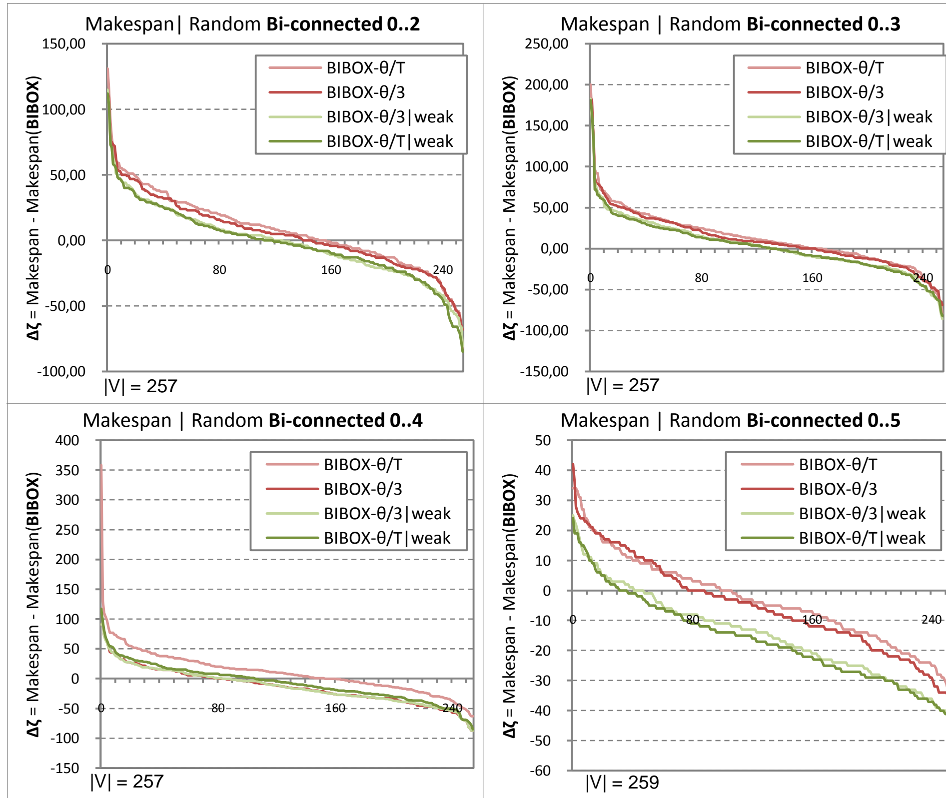


Figure 12. *An evaluation of the benefit of the use of* ***weak special cases*** *instead of the standard ones.* Four variants of the *BIBOX-θ* algorithm are compared: *BIBOX-θ*/T (the standard transposition case is used preferably), *BIBOX-θ*/3(the standard 3-cycle rotation case is used preferably), *BIBOX-θ*/T/weak (the weak transposition case is used preferably), and *BIBOX-θ*/3/weak (the weak 3-cycle rotation case is used preferably). The **difference** of the makespan of solution produced by these algorithms from those produced by the *BIBOX* algorithm is shown (values below zero indicate that the tested algorithm was better than *BIBOX*). Four random bi-connected graphs with the increasing number of unoccupied vertices are used; they have handles of lengths with uniform distribution of ranges: 0..2, 0..3, 0..4, and 0..5 respecitvely. To make the difference visible, results for individual algorithms are sorted in descending order.

This phase transition seems to depend on the average size of handles – for the smaller size of handles the ratio of the number of agents to the number of vertices characterizing this phase transition tends to be higher. The *WHCA** algorithm

generates better solutions than *BIBOX* in most cases (the ratio between the makespan of *BIBOX* and *WHCA** is from 0.5 to 3.0). However, *WHCA** manages to do so only on sparsely occupied environments (number of unoccupied vertices more than 90%). As *WHCA** generates near optimal solutions with respect to the makespan we also have certain indication how far from the optimum solutions generated by *BIBOX* algorithms are. Let us note, that the most difficult instance from our test suite took *WHCA** approximately 2.0 seconds on the runtime configuration (80 agents in the 32×32 grid).

The *BIBOX/MIT* algorithm exhibits performance influenced by the size of the initial θ-like graph. The larger is the graph the worse is the performance of the *BIBOX/MIT* algorithm. This behavior can be observed from the results shown in Figure 10 and Figure 11 using the fact that the longer handles induce larger initial θ-like graph. Grid graphs represent the extreme case – almost all the handles are of the size 1. Both algorithms – *BIBOX* as well as *BIBOX/MIT* – generate solutions of the very similar makespan (the only difference is observable in the case of grid 8×8 with low occupation where *BIBOX/MIT* is marginally better).

Regarding the makespan, the *BIBOX* style solving process represents the better alternative than *MIT* when at least two unoccupied vertices are provided.

An interesting question is whether the use of optimal solutions to weak cases instead of standard ones does really help. Results reported in Figure 12 are devoted to this question. A comparison of the *BIBOX* algorithm with the variants of the *BIBOX-θ* algorithm is shown.

Four variants of the *BIBOX-θ* algorithm are compared: *BIBOX-θ*/T (the standard transposition case is used preferably), *BIBOX-θ*/3 (the standard 3-cycle rotation case is used preferably), *BIBOX-θ*/T/weak (the weak transposition case is used preferably), and *BIBOX-θ*/3/weak (the weak 3-cycle rotation case is used preferably). Notice, that the variant presented in the pseudo-code as Algorithm 3 prefers standard transposition cases. If the transposition case is not possible to apply, the corresponding 3-cycle rotation case is used instead (which is always possible). Other variants implement the preference in the analogical way.

The comparison in Figure 12 shows difference of the makespan of solution generated by mentioned three variants of *BIBOX-θ* from the makespan of the corresponding solution generated by the standard *BIBOX* (negative values of the difference indicate that *BIBOX* generated solution with the greater makespan). Four random bi-connected graphs were used for the experiment; the number of vertices was up to 259 (again, the graph had been grown by addition of handles until the size of 256 vertices had been reached). The length of the initial cycle and handles has been selected randomly with the uniform distribution of ranges: 0..2, 0..3, 0..4, and 0..5, respectively. The relatively small ranges are used in order to be able to calculate all the optimal solutions of the special cases in the timeout of 8.0. The size of the θ-like graph, on that special cases appear, directly corres-

ponds to the length of the initial cycle and handles of the handle decomposition. Makespans have been collected for instances with 2 to $|V| - 1$ unoccupied vertices for each graph $G = (V, E)$. To make differences among performances of tested algorithms clearly visible, the difference in makespans has been sorted in the descending order. The difference in makespan tends to be greater for instances with few unoccupied vertices (hence, it is expected that these makespans are sorted to the left or to the right margin in each plot).

Results shown in Figure 12 can be interpreted as that solutions with the smallest makespan are produced by *BIBOX-θ*/T/weak closely followed by *BIBOX-θ*/3/weak. Hence, it is possible to conclude that the use of optimal solutions to weak special cases is beneficial. Moreover, a solution to a weak special is easier to generate since it is less *constrained* than the solution of the corresponding standard case.

Since values of the makespan differences deviate from the uniform distribution around 0 marginally, it is also possible to conclude that variants of *BIBOX-θ* does not improve the makespan significantly in comparison with *BIBOX* on instances with at least two unoccupied vertices. Thus, the use of *BIBOX-θ* is substantiated only for instances with just a single unoccupied vertex (where the *BIBOX* algorithm is not applicable).

4.2. **Parallelism Evaluation**

The exact meaning of the term *parallelism* is the value obtained as the ratio of the total number of moves divided by the makespan. The result is the average number of moves performed at each time step. High parallelism is typically desirable since it implies the smaller makespan.

In the experiments, we observed how the average parallelism changes while the number of unoccupied vertices is increasing. The same set of setups as in the case of makespan evaluation was used. Results regarding bi-connected graphs are shown in Figure 13 results regarding grids are shown in Figure 14. The parallelism-increasing algorithm [27] was used to post-process the solutions. In case of WHCA* the initial solution was already parallel but in the sense of PMG; we parallelized it further according to pCPF (which however made almost no change as in instances solvable by WHCA* agents were rather isolated).

On random bi-connected graphs, the parallelism of solutions slightly increases as the number of unoccupied vertices reaches approximately 50% occupancy. This behavior is yet more expressed on the grid graphs. The increase of the parallelism is steeper in this case. When the number of unoccupied vertices is higher than some threshold a different behavior can be observed. The fewer agents are in the graph the lower is the parallelism. It can be also observed that parallelism correlates with the average length of handles of the handle decomposition – this is caused by the fact that all the agents in the handle are moving at once. Another

characteristic, which the parallelism correlates with, is the *diameter* [33] of the graph. This correlation can be observed on tests with grid graphs in Figure 14. The reason for this correlation is the fact that all the agents along a path connecting two vertices in the graph moves at once when the unoccupied vertex is relocated. The average length of such paths correlates with the diameter of the graph.

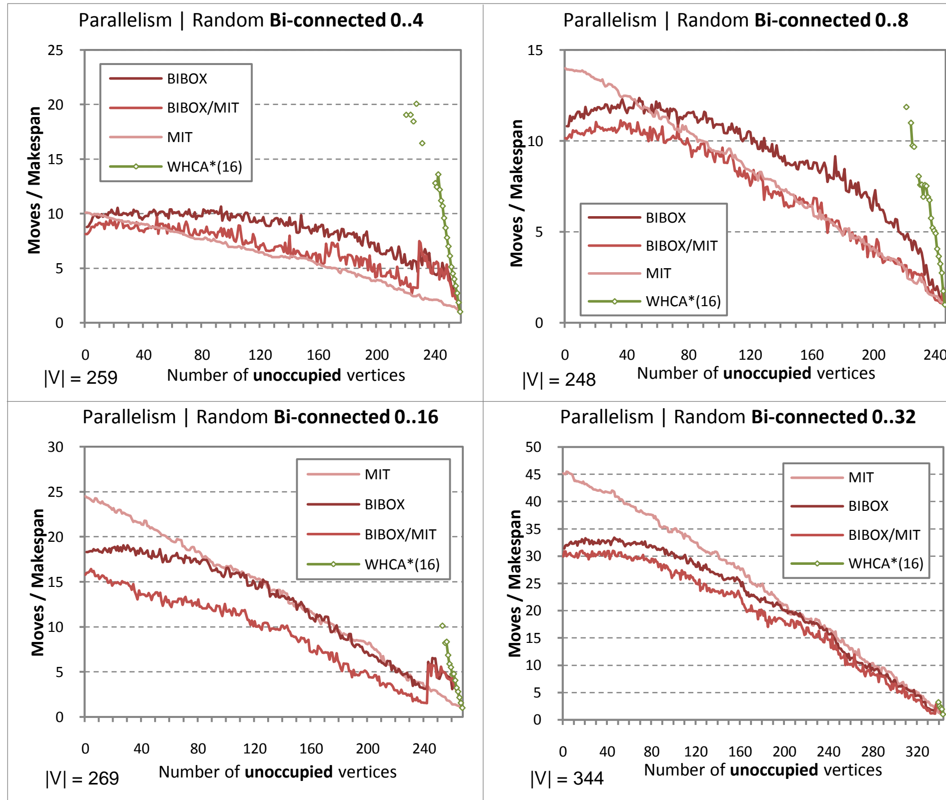


Figure 13. ***Average parallelism*** *of solutions generated by tested algorithms for instances over* ***random bi-connected*** *graphs. BIBOX*, *BIBOX/MIT*, *MIT*, and *WHCA** are compared. Four random bi-connected graphs were used – random lengths of initial cycle and handles of the handle decomposition have uniform distribution of the range: 0..4, 0..8, 0..16, and 0..32. The average parallelism is the total number of moves divided by the makespan.

Regarding the *MIT* algorithm, it can be observed that the parallelism of its solutions decreases almost linearly with the increasing number of unoccupied vertices. Without providing further details, the explanation of this behavior is that all the phases of the algorithm are rather homogenous. Thus, as occurrence of agents is getting linearly sparser the parallelism decreases almost linearly. Recall, that the *BIBOX* algorithm behaves differently. All the movements take place in the unfinished part of the graph only, which is relatively getting smaller as the *BIBOX* algorithm proceeds.

Generally, it can be concluded from Figure 13 and Figure 14 that solutions generated by the *BIBOX* and *BIBOX/MIT* algorithms allow higher parallelism than that of *MIT*. Consequently, it can be observed together from Figure 10, Figure 11, Figure 13, and Figure 14 that the total number of moves, which solutions generated by *BIBOX* and *BIBOX/MIT* consist of, are still order of magnitude smaller than that of *MIT*. Thus, the performance of the *BIBOX* algorithms is not caused by the higher parallelism but also by the smaller size of the generated sequential solutions.

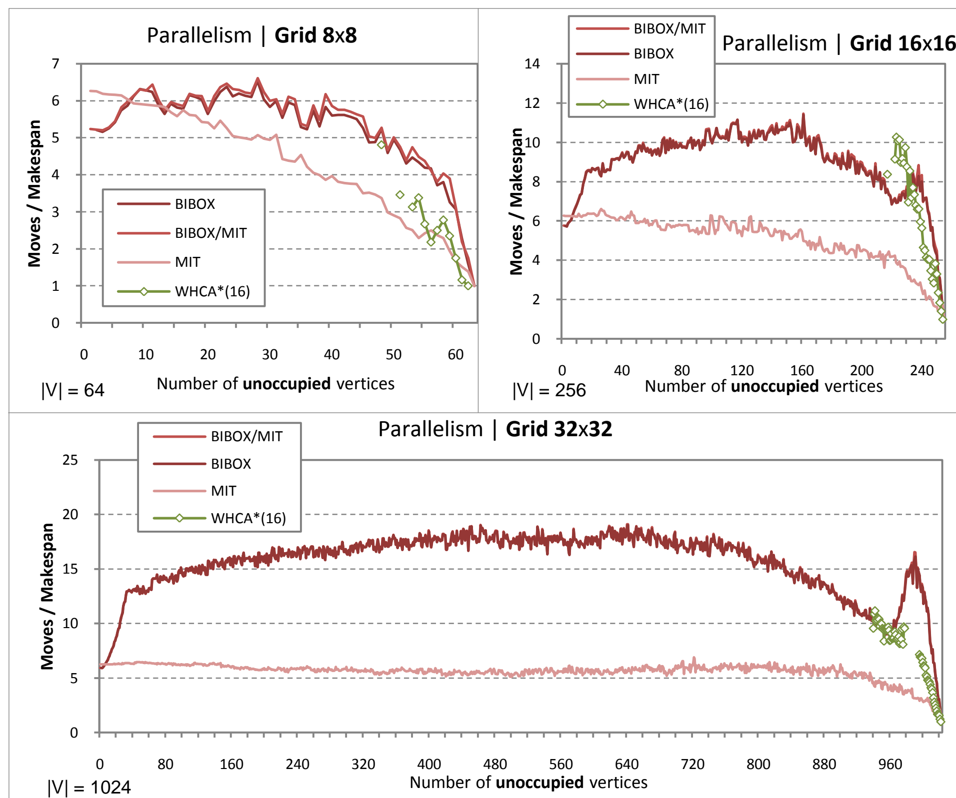


Figure 14. ***Average parallelism*** *comparison of solutions of instances over* ***square grids****. BIBOX, BIBOX/MIT, MIT*, and *WHCA** are compared on three grids: 8 × 8, 16 × 16, and 32 × 32.

Results regarding WHCA* indicate that typically all the agents move. The explanation is that the no-op (that is, an agent does not move) is chosen only if it is necessary to avoid another agent, which is relatively rare situation. Otherwise a move through that an agent can approach its goal is chosen. On random bi-connected graphs WHCA* tends to reach higher parallelism than the other tested algorithms. On grid it seems that no simple statement can be done.

The development of the number of movements per time step called *step parallelism* is shown in Figure 15. This experiment was done with the *BIBOX* algorithm only on a random bi-connected graph where lengths of the initial cycle and handles were randomly selected with the uniform distribution with of the range 0..4. There were exactly two unoccupied vertices in the input graph.

Peaks in Figure 15 correspond to parallel movements along long paths. The density and height of peaks is getting slightly smaller as the algorithm proceeds. This is caused by the fact that the part of the graph affected by movements is getting smaller. Other values correspond to various rotations along cycles are done intensively by the algorithm. The absolute number of parallel movements corresponding to these rotations does not change as the algorithm proceeds (the average size of a cycle in the unfinished part of the graph is still the same since the graph was generated uniformly).

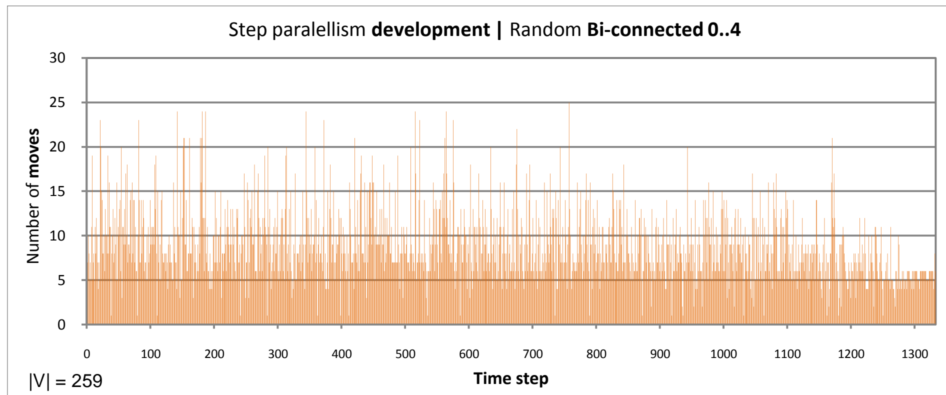


Figure 15. ***Step parallelism development*** *of in a solution generated by BIBOX.* The random bi-connected graph was generated with the length of the initial cycle and handles having uniform distribution of the range 0..4. There were exactly two unoccupied vertices. The development of the *step parallelism* (number of moves per time step) over time is shown.

4.3. **Scalability Evaluation**

Scalability tests were aimed on the makespan of generated solution and the overall runtime necessary to produce the parallel solution while the number of unoccupied vertices was fixed to 2 and the size of the graph was varying. The overall time is the time necessary to produce a sequential solution plus the time needed to increase its parallelism.

The *BIBOX*, *BIBOX/MIT*, *BIBOX-θ*/T/weak, *BIBOX-θ*/3/weak, and *MIT* were compared. Algorithms *BIBOX-θ*/T and *BIBOX-θ*/3 were ruled out since they are outperformed by *BIBOX-θ*/T/weak and *BIBOX-θ*/3/weak respectively as it has been shown in Section 4.1. *BIBOX-θ*/T/weak and *BIBOX-θ*/3/weak are slightly faster supposed that all the records in the database of optimal solutions are pre-

computed off-line (the shorter resulting solution needs to be produced than in the case of *BIBOX-θ*/T and *BIBOX-θ*/3). However, they are significantly faster if the optimal solutions need to be computed on-line (on demand) [21, 22] as the optimal solution to weak special case is easier to find than the optimal solution to the standard special case.

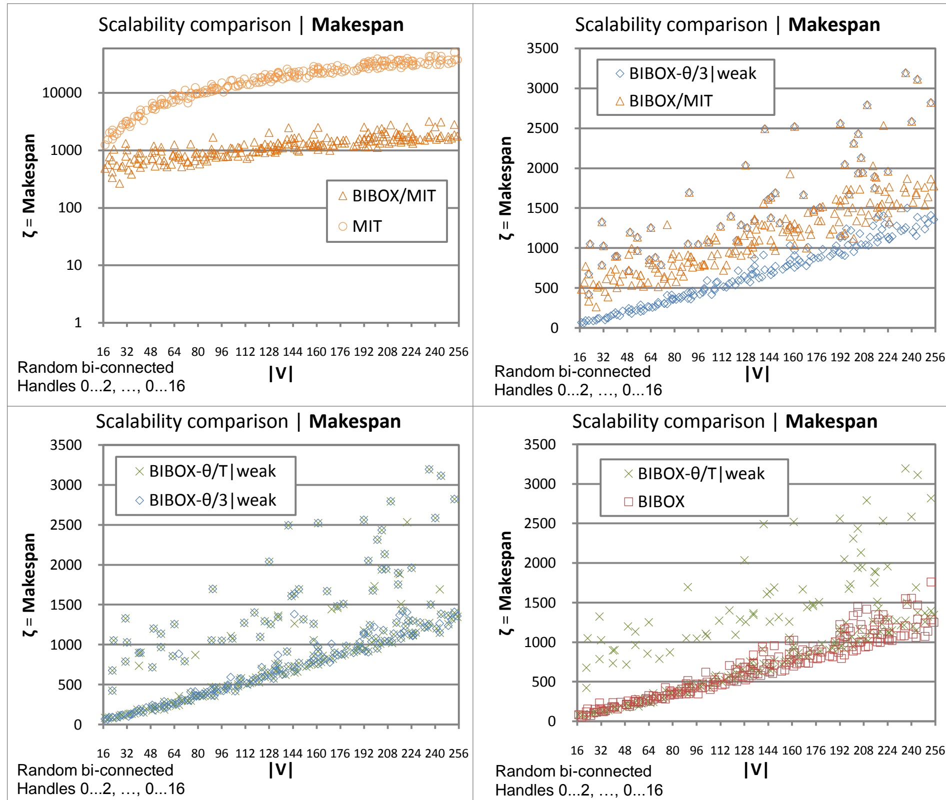


Figure 16. *A* ***comparison of the scalability*** *of tested algorithms with respect to the* ***makespan***. Five algorithms were compared: *BIBOX*, *BIBOX/MIT*, *BIBOX-θ*/T/weak, *BIBOX-θ*/3/weak, and *MIT*. Approximately 250 pCPF instances over various random bi-connected graphs containing 16 to 256 vertices were used. The range of the uniform distribution of lengths of handles in the random generation was: 0..2, …, 0..16. Algorithms are sorted from left/top to right/bottom according to the increasing performance (*MIT* – worst; *BIBOX* - best). Each sub-plot shows the relative comparison of two algorithms.

Tests targeted on scalability used the different setup of instances of pCPF than previous tests. Now, approximately 250 instances on bi-connected graphs with the size varying from 16 to 256 vertices were generated. Random lengths of the initial cycle and handles of the handle decomposition were selected randomly from uniform distribution with ranges: 0..2,…, 0..16. Such selection guarantees

that graphs with short handles as well as graphs with long handles are included. There were exactly two unoccupied vertices in all the tested instances.

Scalability evaluation for the makespan is shown in Figure 16. The makespan for the increasing number of vertices is shown. Experiments in Figure 17 used the same setup (the same set of instances); the difference from Figure 16 is just that the runtime is shown. In both figures, algorithms are compared pair-wise from the worst performing to the best performing pair (the pair of algorithms that are closest to each other according to the given characteristic is compared).

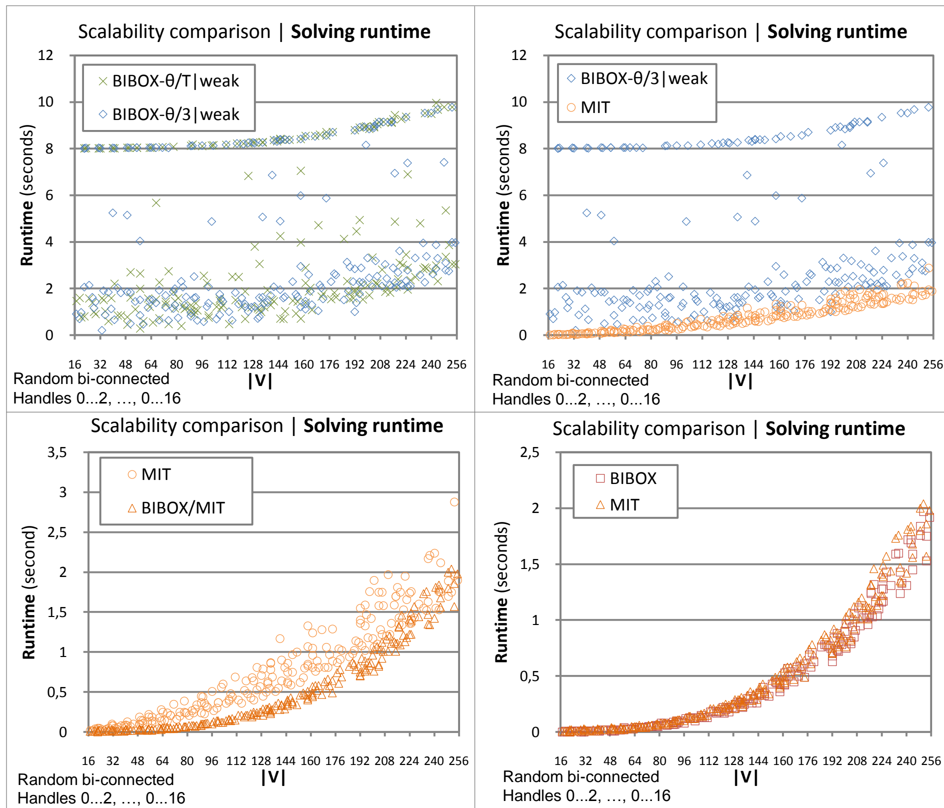


Figure 17. *A* ***comparison of the scalability*** *of tested algorithms with respect to the* ***runtime****. BIBOX*, *BIBOX/MIT*, *BIBOX-θ*/T/weak, *BIBOX-θ*/3/weak, and *MIT* were compared. The setup of instances is the same as for the experiment from Figure 16. Algorithms are sorted from left/top to right/bottom according to the increasing performance. The runtime (the total time necessary to produce sequential solution plus the time for making it parallel) is shown. The runtime increases for the increasing size of the instance (number of vertices).

Results regarding makespan show that the *MIT* algorithm performs as worst while the standard *BIBOX* algorithm produces the best solutions. *BIBOX/MIT*, *BIBOX-θ*/T/weak and *BIBOX-θ*/3/weak are somewhere in the middle. The ma-

kespan of solutions generated by *BIBOX-θ*/T/weak and *BIBOX-θ*/3/weak sometimes jumps up to the makespan of the corresponding solution generated by *BIBOX/MIT*. This happens if *BIBOX-θ*/T/weak or *BIBOX-θ*/3/weak do not manage to compute optimal solution to the special case in the given timeout of 8.0 seconds. In such a case *BIBOX-θ*/T/weak and *BIBOX-θ*/3/weak produces exactly the same solution as *BIBOX/MIT* since they have to switch to the *MIT* mode of generating (sub-optimal) solutions to special cases.

A quite surprising result is that even though *BIBOX-θ*/T/weak and *BIBOX-θ*/3/weak compose the resulting solution over the θ-like graph consisting of the initial cycle and the first handle from the optimal solutions to special cases, it still has the worse makespan than the corresponding solution generated using agents exchanges by the *BIBOX* algorithm. Hence, the second unoccupied vertex has the significant impact on simplifying the solving process.

Results regarding the overall runtime of tested algorithms generally show that *BIBOX-θ*/T/weak and *BIBOX-θ*/3/weak are as slow as the given timeout for computing optimal solutions to the special cases. The more interesting situation is with *MIT*, *BIBOX/MIT*, and *BIBOX* since they have very similar runtimes. The *BIBOX/MIT* tends to be faster than *MIT* while there is only marginal difference between *BIBOX* and *MIT* on larger graphs in favor of *BIBOX*. Observe that the runtime does not exactly correspond to the length of the generated solutions. In other words, certain computations used by *BIBOX* are more time consuming than that of *MIT* (for example *BIBOX* extensively searched for a path when agent is moved).

5. Conclusion and Future Work

Two new algorithms – called *BIBOX* and *BIBOX-θ* – for solving the abstract multi-agent cooperative path-finding with special regard on parallelism (so called pCPFs) were described in this work. Both algorithms are designed for the case when environment is modeled as a bi-connected graph and is densely occupied by agents. Several modified variants of the *BIBOX-θ* algorithm were described as well.

The precise theoretical foundation and experimental analysis of these algorithms is provided. The theoretical foundation is targeted on correctness of the design of algorithms. The experimental analysis is primarily targeted on comparison with the *MIT* algorithm that employs permutation group theory and is capable of solving pCPF instances characterized by the small unoccupied space. To provide the complete image with respect to the related works in cooperative path-finding the comparison with the WHCA* algorithm, which is one of the most commonly used benchmark algorithm for CPF, is given as well.

Although the *MIT* algorithm has promising theoretical properties it has been outperformed by *BIBOX* in terms of the makespan by the order of one to two

magnitudes. Although the asymptotic estimation for the makespan is the same for both *BIBOX* and *MIT*, the multiplication factor in the estimation in the case of *BIBOX* is smaller. Regarding the runtime, *BIBOX* algorithm is slightly faster than *MIT*, which itself is relatively fast (instances with graphs of hundreds of vertices occupied by hundreds of agents are solved within seconds on today's commodity hardware).

The minor drawback of the *BIBOX* algorithm is that it is not able to solve instances of pCPF with just a single unoccupied vertex. This issue has been addressed in this work by proposing modified algorithm called *BIBOX-θ* and its variants called *BIBOX/MIT*, *BIBOX-θ*/T, *BIBOX-θ*/3, *BIBOX-θ*/T/weak, and *BIBOX-θ*/3/weak. They use a different approach to solve the situation on the simple bi-connected graphs consisting of one cycle and one handle connected to it – called θ-like graphs. Except the first mentioned algorithm, all the other algorithms use the database with optimal solutions to special instances over these θ-like graphs – called special cases – of which solutions to all the instances over θ-like graphs can be composed.

Regarding the makespan, all the alternative algorithms outperform *MIT*. If the database of optimal solutions is available in advance, then *BIBOX-θ* algorithms almost match the performance of *MIT* in terms of runtime. If the required optimal solutions to special cases are not available, they need to be computed on-line which is difficult. It can cause a significant slowdown of the algorithm.

Notice, that the performance of both presented algorithms depends on the handle decomposition of the input graph. An interesting question is how to optimize handle decomposition in order to improve makespan or runtime. Is it better to use a small number of large handles or a large number of small handles? This question is out of the scope of this work and it is left for future work.

A considerable drawback of presented algorithms is their limitation on bi-connected graph. Notice, that search-based techniques like WHCA* are not limited to any special class of graphs. Hence, extension of presented algorithms to the general case is of interest. One of the possible approaches is to decompose a given general graph into the tree of bi-connected components [33, 34]. Any of the presented algorithms for bi-connected case can be used over the individual bi-connected components. However, agents need first to be relocated to the target bi-connected components. It may happen that an agent needs to go to the neighboring bi-connected component different from that where it is currently located. If the *bridge* connecting these components is longer than the number of unoccupied vertices then the relocation of the agent will not be possible. Hence, there will be relatively many unsolvable instances in the general case.

Regarding future work, it is also interesting to resolve the question whether optimal solutions of pCPF can be approximated by a (pseudo-) polynomial time algorithm. If an approximation algorithm with (pseudo-) polynomial time com-

plexity is available, it is possible to estimate how far the current solution is from the optimal one even for large and densely occupied instances (currently we have only intuition for sparsely populated instances thanks to experiments with WHCA*). Some study of this kind of approximation algorithms for the special case of $(N^2 - 1)$-puzzle has been done in [11, 12, 13].

Another interesting topic for future work is to study how solutions generated by presented algorithm can be improved. A first view work has been already done in [25]. It is based on identifying and eliminating redundancies from solutions. The performed experiments showed that it is a promising technique.

References

1. T. H. **Cormen**, C. E. **Leiserson**, R. L. **Rivest**, and C. **Stein**. *Introduction to Algorithms (Second edition)*, MIT Press and McGraw-Hill, 2001, ISBN 0-262-03293-7.
2. J. D. **Dixon** and B. **Mortimer**. *Permutation Groups*. Graduate Texts in Mathematics, Volume 163, Springer, 1996, ISBN 978-0-387-94599-6.
3. M. R. **Garey** and D. S. **Johnson**. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1979, ISBN: 978-0716710455.
4. J. E. **Hopcroft**, R. **Motwani**, J. D. **Ullman**. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 2000, ISBN: 978-0201441246.
5. E. **Hordern**. *Sliding Piece Puzzles.* Oxford University Press, 1986, ISBN: 978-0198532040.
6. M. R. **Jansen** and N. R. **Sturtevant**. *Direction maps for cooperative pathfinding.* Proceedings of (AIIDE 2008), pp..AAAI Press, 2008.
7. M. R. **Jansen** and N. R. **Sturtevant**. *A new approach to cooperative pathfinding.* Proceedings of AAMAS 2008, pp. 1401 - 1404, 2008.
8. D. **Kornhauser**, G. L. **Miller**, and P. G. **Spirakis**. *Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications*. Proceedings of the 25th Annual Symposium on Foundations of Computer Science (FOCS 1984), pp. 241-250, IEEE Press, 1984.
9. R. **Luna**, K. E. **Berkis**. *Push-and-Swap: Fast Co-operative Path-Finding with Completeness Guarantees*. Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011), pp. 294-300, IJCAI/AAAI Press, 2011.
10. C. H. **Papadimitriou**, P. **Raghavan**, M. **Sudan**, and H. **Tamaki**. *Motion Planning on a Graph.* Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS 1994), pp. 511-520, IEEE Press, 1994.
11. I. **Parberry**. *A Real-Time Algorithm for the $(n^2$-1)-Puzzle.* Information Processing Letters, Volume 56(1),pp. 23-28, Elsevier,1995.

12. D. **Ratner** and M. K. **Warmuth**. *Finding a Shortest Solution for the N×N Extension of the 15-PUZZLE Is Intractable*. Proceedings of the 5th National Conference on Artificial Intelligence (AAAI 1986), pp. 168-172, Morgan Kaufmann Publishers, 1986.
13. D. **Ratner** and M. K. **Warmuth**. *N×N Puzzle and Related Relocation Problems.* Journal of Symbolic Computation, Volume 10 (2), pp. 111-138, Elsevier, 1990.
14. S. **Russell** and P. **Norvig**. *Artificial Intelligence: A Modern Approach (second edition).* Prentice Hall, 2003, ISBN: 978-0137903955.
15. M. R. K. **Ryan**. *Graph Decomposition for Efficient Multi-Robot Path-Planning*. Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 2003-2008, IJCAI Conference, 2007.
16. M. R. K. **Ryan**. *Exploiting Subgraph Structure in Multi-Robot Path-Planning*. Journal of Artificial Intelligence Research (JAIR), Volume 31, pp. 497-542, AAAI Press, 2008.
17. P. E. **Schupp** and R. C. **Lyndon**. *Combinatorial group theory*. Springer, 2001, ISBN 978-3-540-41158-1.
18. D. **Silver**. *Cooperative Pathfinding*. Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2005), pp. 117-122, AAAI Press.
19. T. **Standley**. *Finding Optimal Solutions to Cooperative Pathfinding Problems.* Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010), pp. 173-178, AAAI Press, 2010.
20. P. **Surynek**. *A Novel Approach to Path Planning for Multiple Robots in Bi-connected Graphs*. Proceedings of the 2009 IEEE International Conference on Robotics and Automation (ICRA 2009), pp. 3613-3619, IEEE Press, 2009.
21. P. **Surynek**. *Towards Shorter Solutions for Problems of Path Planning for Multiple Robots in θ-like Environments*. Proceedings of the 22nd International FLAIRS Conference (FLAIRS 2009), pp. 207-212, AAAI Press, 2009.
22. P. **Surynek**. *Making Solutions of Multi-Robot Path-Planning Problems Shorter Using Weak Transpositions and Critical Path Parallelism*. Proceedings of the 2009 International Symposium on Combinatorial Search (SoCS 2009), University of Southern California, 2009, http://www.search-conference.org/index.php/Main/SOCS09 [July 2009].
23. P. **Surynek**. *An Application of Pebble Motion on Graphs to Abstract Multi-robot Path-planning*. Proceedings of the 21st International Conference on Tools with Artificial Intelligence (ICTAI 2009), pp. 151-158, IEEE Press, 2009.
24. P. **Surynek**. *An Optimization Variant of Multi-Robot Path-Planning is Intractable.* Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010), pp. 1261-1263, AAAI Press, 2010.
25. P. **Surynek** and P. **Koupý**.*Improving Solutions of Problems of Motion on Graphs by Redundancy Elimination.* Proceedings of the ECAI 2010 Workshop on Spatio-Temporal Dynamics (ECAI STeDy 2010), pp. 37-42, University of Bremen, 2010.

26. P. **Surynek**. *Abstract Path Planning for Multiple Robots: A Theoretical Study.* Technical Report, ITI Series, 2010-503, http://iti.mff.cuni.cz/series/index.html, Institute for Theoretical Computer Science, Charles University in Prague, Czech Republic, 2010.

27. P. **Surynek**. *Abstract Path Planning for Multiple Robots: An Empirical Study.* Technical Report, ITI Series, 2010-504, http://iti.mff.cuni.cz/series/index.html, Institute for Theoretical Computer Science, Charles University in Prague, Czech Republic, 2010.

28. R. E. **Tarjan**. *Depth-First Search and Linear Graph Algorithms.* SIAM Journal on Computing, Volume 1 (2), pp. 146-160, Society for Industrial and Applied Mathematics, 1972.

29. K. C. **Wang** and A.**Botea**. *Tractable Cooperative path-finding on Grid Maps*. Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009), pp. 1870-1875, IJCAI Conference, 2009.

30. K. C. **Wang**. *Bridging the Gap between Centralised and Decentralised Multi-Agent Pathfinding.* Proceedings of the 14th Annual AAAI/SIGART Doctoral Consortium (AAAI-DC 2009), pp. 23-24, AAAI Press, 2009.

31. K. C. **Wang** and A.**Botea**. *Fast and Memory-Efficient Multi-Agent Pathfinding.* Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008), Australia, pp. 380-387, AAAI Press, 2008, ISBN 978-1-57735-386-7.

32. K. C. **Wang** and A. **Botea**. *Scalable Multi-Agent Pathfinding on Grid Maps with Tractability and Completeness Guarantees*. Proceedings of the European Conference on Artificial Intelligence (ECAI 2010), IOS Press, 2010.

33. D. B. **West**. *Introduction to Graph Theory.* Prentice Hall, 2000, ISBN: 978-0130144003.

34. J. **Westbrook**, R. E. **Tarjan**. *Maintaining bridge-connected and biconnected components on-line*. Algorithmica, Volume 7, Number 5&6, pp. 433–464, Springer, 1992.

35. R. M. **Wilson**. *Graph Puzzles, Homotopy, and the Alternating Group.* Journal of Combinatorial Theory, Ser. B 16, pp. 86-96, Elsevier, 1974.

Appendix

Lemma 4 *(soundness of Move-Agent)*. If an original location of an agent a, a goal location v, and an unoccupied vertex are all located in the same unlocked bi-connected component of the graph G, then the procedure *Move-Agent* correctly moves the agent a from its original location to v. ■

Proof. Recall how the procedure *Move-Agent* works. First, a path $\varphi = [w_1^\varphi, w_2^\varphi, \ldots, w_{j_\varphi}^\varphi]$ connecting $S_A(a)$ and v that is contained in the same bi-connected component is found. The path φ is then traversed while the agent a is moved along its edges.

The proof of soundness will proceed as mathematical induction according to the number of edges of φ already traversed. In all the steps, the agent a and the unoccupied vertex should be located in the bi-connected component containing φ. Initially, this condition holds. Consider that an agent a is located in w_i^φ for $i \in \{1,2, \ldots, j_\varphi\}$ and need to be moved to w_{i+1}^φ. The vertex w_i^φ is locked and w_{i+1}^φ is made unoccupied. To make w_{i+1}^φ unoccupied an unlocked path connecting the original location of the unoccupied vertex and w_{i+1}^φ must exist in the bi-connected component. Since it is supposed that w_i^φ, w_{i+1}^φ, and the unoccupied vertex are all in the same bi-connected component the alternative path connecting w_{i+1}^φ and the unoccupied vertex in this bi-connected component avoiding w_i^φ must exist (since otherwise removal of w_i^φ would make the bi-connected component disconnected which is a contradiction). This path is used to transfer the unoccupied vertex to w_{i+1}^φ. Having w_{i+1}^φ unoccupied the vertex w_i^φ is unlocked and a is moved to w_{i+1}^φ along the edge $\{w_i^\varphi, w_{i+1}\}$. After this step, the required condition holds again (a supporting illustration is shown in Figure 4). ■

Lemma 5 ***(soundness of Exchange-Agents).*** If the arrangement of agents within the cycle C_0 is regarded as a permutation, then the output arrangement produced by the procedure *Exchange-Agents* corresponds to a permutation where the input agents a and b are transposed with respect to the permutation corresponding to the input arrangement. ■

Proof. It is needed to check whether the orderings of agents between a and b and between b and a (with respect to the positive orientation of the cycle) remain unchanged while a and b are transposed. This is done using detailed case analysis of what happens. Let $C_0 = [w_1^0, w_2^0, \ldots, w_l^0]$, then there are $l - 2$ agents located in C_0 at the moment before the cycle is rotated positively (situation at line 9 of *Exchange-Agents* - see stage (i) in Figure 18). The agent a is already stored in v and the two unoccupied vertices are u and $next_\circlearrowleft(C_0, u)$. Let agents occupying vertices of the cycle in the interval between $\Phi_A(b)$ and u with respect to the positive orientation (excluding boundaries) are denoted $b_1, b_2, \ldots, b_k$ respectively; let agents occupying vertices of the cycle in the interval between $next_\circlearrowleft(C_0, u)$ and $\Phi_A(b)$ with respect to the positive orientation (again excluding boundaries) are denoted as $a_1, a_2, \ldots, a_{l-k-3}$. The series of ρ positive rotation of C_0 follows to move the agent b into $next_\circlearrowleft(C_0, u)$ (see stage (ii) in Figure 18). Now, all the agents $b_1, b_2, \ldots, b_k$, $a_1, a_2, \ldots, a_{l-k-3}$, and b are ρ steps forward with respect to

their location before the series of rotations. Then the second unoccupied vertex (other than u) is moved in the positive direction towards $prev_{\circlearrowright}(C_0, u)$ (recall, that the movement in the negative direction is not possible, since u is locked at the moment - see stage (iii) in Figure 18). Next, agents are exchanged: that is, b is moved to v and a is moved to $prev_{\circlearrowright}(C_0, u)$ (see stage (iv) in Figure 18 and lines 14-17 of *Exchange-Agents*). At this step, agents $b_1, b_2,\ldots, b_k$ are ρ steps forward with respect to their location before the series of rotations; agent $a_1, a_2,\ldots, a_{l-k-3}$ are $\rho - 1$ forwards with respect to their location before the series of rotations (the difference is caused by the fact that unoccupied vertex went through agents $a_1, a_2,\ldots, a_{l-k-3}$ but not through agents $b_1, b_2,\ldots, b_k$). Finally, the agent a is $\rho - 1$ steps forward with respect to the location of b before the series of rotations.

The series of ρ rotation in the negative direction places agents $b_1, b_2,\ldots, b_k$ to their original positions; agents $a_1, a_2,\ldots, a_{l-k-3}$ are placed 1 step backward with respect to their original position before rotations, and a is 1 step backward with respect to the original position of b before the series of rotations (see stage (v) in Figure 18). This inconsistency however, is caused by a different location of the second unoccupied vertex which now between a and b_1 with respect to the positive orientation of the cycle (this was not the case in the original arrangement before rotations).

To see that the transposition of a and b has been really obtained, the movement of the second unoccupied vertex into $next_{\circlearrowright}(C_0, u)$ in the negative direction can be done. This moves agents $a_1, a_2,\ldots, a_{l-k-3}$ to their original positions before rotations and the agent a to the original position of b (see stage (vi) in Figure 18). As this is a step used only for purposes of the proof, the algorithm actually does not perform it. ■

Proposition 6 *(BIBOX - soundness and completeness).* The *BIBOX* algorithm always terminates and produces a solution of a given input instance of pCPF $\Sigma = (G = (V, E), A, S_A^0, S_A^+)$. ■

Proof. To verify soundness and completeness of the *BIBOX* algorithm it is necessary to check preconditions of each operation performed in the course of its execution. This is a trivial task in almost all the cases except the case of searching for a path satisfying certain conditions. This issue concerns the search for vertex disjoint paths φ and χ within the main function *BIBOX-Solve* at line 2 and the search for a path connecting a given pair of vertices avoiding the locked ones.

The existence of vertex disjoint paths φ and χ has been already treated by Lemma 2. Thus, it remains to verify that a required unlocked path always exists.

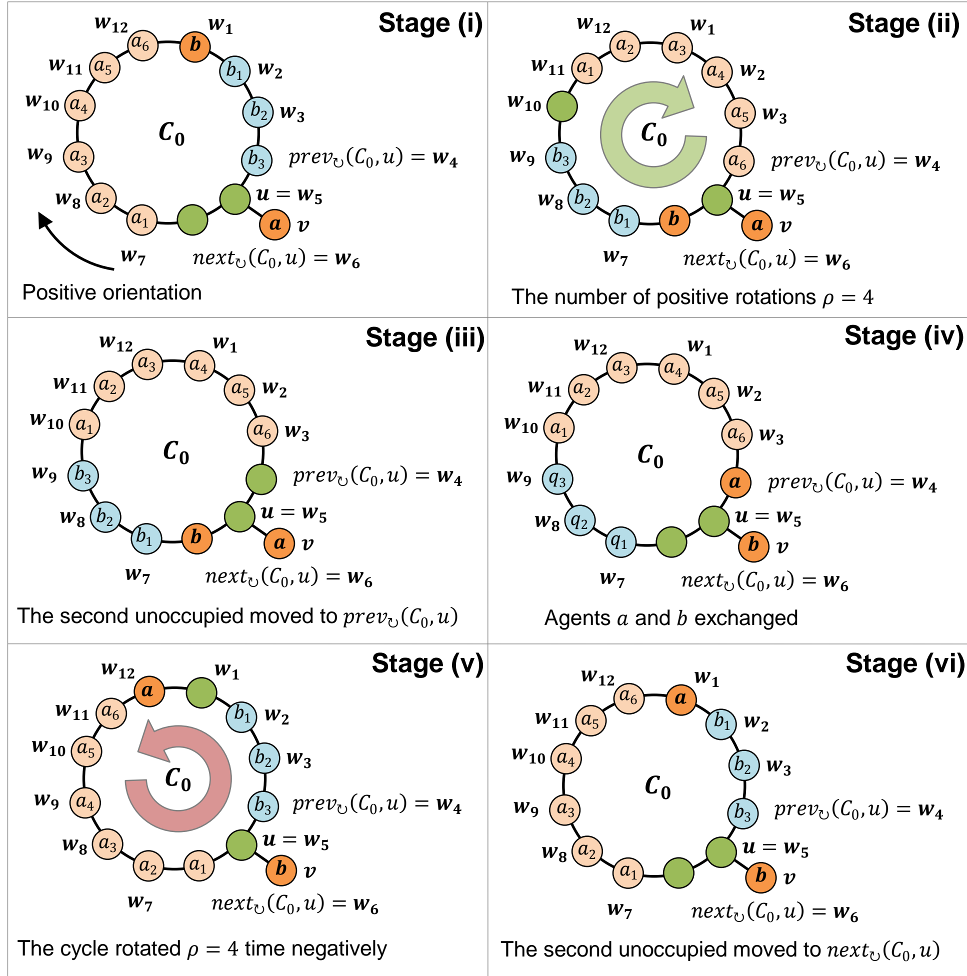


Figure 18. *The* ***progression of the exchange*** *of a pair of agents within an initial cycle of the handle decomposition.* Agents a and b in a cycle consisting of 12 vertices are exchanged while the ordering of other agents within the cycle is preserved. The figure illustrates the progression of the procedure *Exchange-Agents* from line 7 to 20.

A path containing unlocked vertices is constructed within the procedure *Make-Unoccupied* (lines 2-3) which is called by *Solve-Regular-Handle* (lines 3, 5, 12, 16, 26, and 35), *Solve-Original-Cycle* (lines 4, 6, 10, and 12), and *Exchange-Agents* (lines 2, 8, and 13). A pair of vertex disjoint paths containing unlocked vertices constructed within the procedure *Move-Agent* (lines 1-5) which is called by *Solve-Regular-Handle* (lines 10, 24, and 33). All these cases must be examined.

There will be the following invariant within *Solve-Regular-Handle* – at the beginning of every iteration of the loop at line 7, an unoccupied vertex must be located in the not yet solved part of the graph. More precisely, let the bi-connected

subgraph without the internal vertices of the already solved handles be denoted as $G^{\sim}$ and let $G^{\sim}$ without the internal vertices of H_c, where H_c is the currently solved handle by *Solve-Regular-Handle*, be denoted as $G^{'}$ (see Figure 6). Then an unoccupied vertex is needed to be located in $G^{'}$ every time the loop at line 7 starts. The assumption holds at the beginning and it is needed to check if it holds after every iteration of the loop. Furthermore, an invariant that both unoccupied vertices are located in $G^{\sim}$ at the start of *Solve-Regular-Handle* will be also checked. Again, this invariant holds at the beginning.

Vertices w and z which are used as parameters of the call of *Make-Unoccupied* at lines 3 and 5 respectively of *Solve-Regular-Handle* are both in $G^{'}$. Since $G^{\sim}$ is completely unlocked at lines 3 of *Solve-Regular-Handle* and it is assumed that an unoccupied vertex is located in $G^{\sim}$, an unlocked path connecting w and an unoccupied vertex must exist. The construction of a path within the call at line 5 of *Solve-Regular-Handle* must additionally take into account that w is locked. As the subgraph $G^{\sim}$ is bi-connected, it remains connected even if w is removed and hence the path exists.

At line 12 of *Solve-Regular-Handle*, a connection vertex v^c of the currently solved handle H_c is being made unoccupied while internal vertices of H_c and the second connection vertex u^c are locked. According to above invariants and the fact that the call of *Move-Agent* at line 10 does not invalidate them, as it is prevented from using internal vertices of H_c by locking them at line 8, an unoccupied vertex is now located in $G^{'}$ (except u^c). The graph $G^{'}$ is bi-connected and without u^c, which is locked just before, it is all unlocked and still connected. Hence the required path exists.

The call of *Make-Unoccupied* at line 16 of *Solve-Regular-Handle* has the connection vertex u^c of the currently solved handle H_c as a parameter. The subgraph $G^{'}$ is now unlocked and according to invariants an unoccupied vertex is located in $G^{'}$. Since $G^{'}$ is connected, there exists an unlocked path connecting u^c and the unoccupied vertex.

At line 26 of *Solve-Regular-Handle*, the connection vertex u^c of the handle H_c is made unoccupied. The situation is that a vertex y, which is in $G^{'}$ and outside the cycle associated with the current handle H_c, is locked while the rest of $G^{'}$ is unlocked. Again, the unlocked part of the graph corresponds to a bi-connected subgraph $G^{'}$ from which one vertex was removed. Thus, the unlocked part of the graph constitutes a connected component. An unoccupied is also located in the unlocked part. This holds from the invariants and from the fact that movements at lines 20 and 24 cannot relocate it outside $G^{'}$ as *Rotate-Cycle*$^{+}$ does not relocate the input unoccupied vertex and *Move-Agent* cannot go outside the unlocked part which is exactly $G^{'}$ at the moment due to locking of internal vertices of H_c at line 22. Hence, there exists an unlocked path connecting the unoccupied vertex and u^c.

At line 35 of *Solve-Regular-Handle* the task is to make unoccupied a connection vertex v^c of the handle H_c. The situation is again very similar; the unlocked part of the graph is constituted by G' without u^c. Thus, unlocked vertices constitute a connected subgraph. The unoccupied vertex must be located in the unlocked part as it was located in u^c after the execution of line 26 and subsequent movements cannot relocate it outside G' (*Rotate-Cycle*$^-$ at line 29 does not relocate the input unoccupied vertex and *Move-Agent* at line 33 remains in the unlocked part). Thus, there exists an unlocked path connecting the unoccupied vertex and v^c.

An unoccupied vertex is located in G' at the end of the iteration of the loop starting at line 7 since it is v^c in both major execution branches (notice that calls of *Rotate-Cycle*$^+$ at line 14 and 37 respectively preserve positions the unoccupied vertex v^c). Thus, the first invariant holds. Since it is assumed that goal positions of unoccupied vertices are within the initial cycle, no unoccupied vertex can be stored in H_c. Hence, both unoccupied vertices are in G' at the end of the execution of the loop (that is, they are within $G^\sim$ with respect to the processing of next handle).

The soundness of the procedure *Solve-Original-Cycle* is partially implied by the soundness of the procedure *Exchange-Agents* which is treated by Lemma 5. The basic assumption of *Solve-Original-Cycle* is that both unoccupied vertices are located in the original cycle C_0 of the handle decomposition; all the vertices of the graph except C_0 are locked. The assumption directly corresponds to the second invariant preserved along the calls of *Solve-Regular-Handle* within the loop at lines 5-7 of *BIBOX-Solve*.

At line 4 of *Solve-Original-Cycle* a vertex w_1^0 (the first vertex of the cycle with respect to the positive orientation) is being made unoccupied. An unlocked path in the cycle from any of its vertices to w_1^0 exists, hence making w_1^0 unoccupied is possible. The situation at line 6 of *Solve-Original-Cycle* is little bit different; now the vertex w_1^0 is locked and a vertex w_2^0 (the second vertex of C_0 with respect to the positive orientation) is being made unoccupied. Thus, an unlocked path connecting the second unoccupied vertex with w_2^0 is searched. Such path exists since removing w_1^0 from the cycle does not disconnect it. The situation at lines 10 and 12 of *Solve-Original-Cycle* is analogical.

The soundness of the procedure *Move-Agent* itself is treated separately by Lemma 4. However, preconditions of the Lemma 4 need to be checked – that is, whether all the calls of *Move-Agent* moves an agent within the single unlocked bi-connected component and whether the unoccupied vertex is located in the same unlocked bi-connected component as well.

The situation before the call of *Move-Agent* at line 10 of *Solve-Regular-Handle* is that G' is unlocked while the rest of the graph is locked. An unoccupied vertex is located in G' which is ensured by the invariant. The task is to move an

agent $\Phi_A^+(w_i^c)$, which is known to be located in $G^{'}$ (this is, treated by the execution branch at line 9), to the connection vertex u^c of the handle H_c. As the unoccupied vertex and both the agent $\Phi_A^+(w_i^c)$ and u^c are located in $G^{'}$ constituting a bi-connected component, preconditions of Lemma 4 are satisfied.

The call of *Move-Agent* at line 24 of *Solve-Regular-Handle* moves the agent $\Phi_A^+(w_i^c)$ to a vertex y which is located in $G^{'}$ and outside the cycle associated with the handle H_c at the same time. Again, $G^{'}$ is unlocked while the rest of the graph is locked. The agent $\Phi_A^+(w_i^c)$ is known to be located in the connection vertex v^c of H_c and one of the unoccupied vertices is the second connection vertex u^c. Thus, the unlocked vertices constitutes a bi-connected component where the agent $\Phi_A^+(w_i^c)$, the vertex y, and the unoccupied vertex are located. Hence, preconditions of Lemma 4 are satisfied.

Finally, the task of the call of *Move-Agent* at line 33 of *Solve-Regular-Handle* is to move an agent $\Phi_A^+(w_i^c)$ to a connection vertex u^c of the current handle H_c which is assumed to be unoccupied at the moment. It is known that the agent $\Phi_A^+(w_i^c)$ is located in y from the previous case. $G^{'}$ is again unlocked while the rest of the graph is locked. Thus, the agent $\Phi_A^+(w_i^c)$ and the unoccupied vertex u^c are both located in $G^{'}$ which is a bi-connected component. Thus, preconditions of Lemma 4 are satisfied again.

At this point, it is possible to conclude that all the steps of the algorithm are correctly defined. Since the number of successfully placed agents strictly increases as the algorithm proceeds, the algorithm always terminates and produces a solution to the input instance. ■

Proposition 7 ***(BIBOX – worst-case time complexity).*** The **worst-case time** complexity of the *BIBOX* algorithm is $\mathcal{O}(|V|^3)$ with respect to an input pCPF instance $\Sigma = (G = (V, E), A, S_A^0, S_A^+)$. ■

Proof. The construction of a handle decomposition (line 1 of *BIBOX-Solve*) takes $\mathcal{O}(|V| + |E|)$ steps (Lemma 1). The same estimation holds for transforming the goal arrangement of agents (line 2 of *BIBOX-Solve*) and augmenting the final solution (line 9 of *BIBOX-Solve*) according to a pair of vertex disjoint paths φ and χ.

There are at most $|V|$ agents (since $|A| < |V|$) to be placed within handles of a handle decomposition $\mathcal{D} = [C_0, H_1, H_2, \ldots, H_d]$. Placing an agent a within H_c with $c \in \{1,2, \ldots, d\}$ requires at most $|H_c|$ rotations of the cycle $C(H_c)$ in the positive direction (procedure *Rotate-Cycle*$^+$) in the case when a is needed to be moved outside H_c. Then, at most $|H_c|$ rotations of $C(H_c)$ in the negative direction (procedure *Rotate-Cycle*$^-$) are necessary to put agents in H_c back to their original positions; and finally, one rotation of $C(H_c)$ in the positive direction is necessary to get the agent a to its position within H_c. Altogether at most $2|H_c| + 1$ rotations

of $C(H_c)$ are necessary. One rotation of the cycle $C(H_c)$ requires at most $|C(H_c)|$ steps. If the agent a does not need to be moved outside H_c only one positive rotation of $C(H_c)$ is needed. Thus, all the rotations needed to place the agent a consume at most $|C(H_c)| \cdot (2|H_c| + 1)$ steps.

It is also necessary to move the agent a (procedure *Move-Agent*) during the placement operation. There are up to 2 calls of *Move-Agent* per agent placement within the handle H_c. A more careful analysis must be done here since the agent a must be moved along a path of the length up to $|V|$ while non-trivial amount of work needs to be done per each edge traversal.

A vertex in front of the current location of a needs to be made unoccupied every time an edge is traversed by a. Thus a path connecting the unoccupied vertex and the location in front of a must be found while the vertex containing a should be avoided by the path. Agents are then shifted along the found path. The path should be searched in the graph constituted by the initial cycle and handles of the handle decomposition that contains at least one internal vertex. Such a graph contains only linear number of edges with respect to the number of vertices and thus the search for the path can be completed in $O(|V|)$ steps. The subsequent shifting of agents consumes at most $|V|$ steps. Hence, the single traversal of an edge by the agent a requires $O(|V|)$ steps. Altogether, $O(|V|^2) + O(|V| + |E|)$ steps are required by operations for moving of agents.

There are also up to 5 calls of the operation for making some vertex unoccupied (procedure *Make-Unoccupied*) per agent placement. The operation for making some vertex unoccupied requires $O(|V| + |E|)$ steps; this is accounted to the search for a shortest path connecting the original and the goal location. Shifting agents itself along the found path is less consuming; it requires at most $|V|$ steps. Thus, at most $5|V|+ O(|V| + |E|)$ steps are consumed by making vertices unoccupied in course of placing a.

In total, at most $(2|H_c| + 1) \cdot |C(H_c)| + 2|V|^2 + 5|V| + O(|V| + |E|)$ steps are necessary to place a into H_c. Since $|H_c| \leq |C(H_c)| \leq |V|$, the total number of steps is at most $(2|V| + 1) \cdot |V| + 2|V|^2 + 5|V| + O(|V| + |E|)$ which is $O(|V|^2)$.

The remaining operations consume the constant time. Since there are at most $|V|$ agents, the whole process of placing agents into handles takes $O(|V|^3)$ steps.

It remains to analyze the time required by placing agents within the original cycle C_0. Each agent a requires 2 operations of making a vertex unoccupied (the first and the second vertex C_0 are made unoccupied – lines 4 and 6 of *Solve-Original-Cycle*) and at most one operation of exchanging agents. Since the initial and the goal position of both mentioned relocations of the unoccupied vertex are located in C_0, the operation requires only $|C_0|$ steps in the worst-case. The operation of exchanging agents requires at most $2|C_0|$ rotations in the positive direction (lines 5 and 11 of *Exchange-Agents*) and at most $|C_0|$ rotations in the negative

direction (line 19 of *Exchange-Agents*). Next, there are 3 calls of the operation for making some vertex unoccupied (call of the procedure *Make-Unoccupied* at lines 2, 8, and 13). Observe that the unoccupied vertex and the target vertex of the relocation are located in C_0 in all the cases. Thus, each of these operations requires at most $|C_0|$ steps. Altogether, $3|C_0|$ steps are required for making vertices unoccupied during exchanging a pair of agents. The time consumption of the remaining operations performed during a single exchange of agents is constant.

A single exchange of a pair of agents requires at most $3|C_0|^2 + 5|C_0|$ steps in total. Placing all the agents into the original cycle hence consumes at most $|C_0| \cdot (3|C_0|^2 + 5|C_0|)$ steps. Since $|C_0| < |V|$, the total number of steps required for solving the initial cycle is at most $|V| \cdot (3|V|^2 + 5|V|)$ which is $\mathcal{O}(|V|^3)$.

It was shown that the worst-case time of $\mathcal{O}(|V|^3)$ is necessary to solve regular handles as well as the initial cycle thus the worst-case time complexity of the BIBOX algorithm is $\mathcal{O}(|V|^3)$. ■

Using almost the same arguments as in the above proof it is possible to calculate the worst-case makespan of solutions generated by the BIBOX algorithm. Notice that the algorithm generates movement of the agent in almost every step referred in the time complexity analysis.

Proposition 8 ***(BIBOX – makespan of the solution).*** The **worst-case makespan** of the solution produced by the BIBOX algorithm (that is, the number ζ) for an input instance of pCPF $\Sigma = (G = (V, E), A, S_A^0, S_A^+)$ is $\mathcal{O}(|V|^3)$. ■

Pavel **Surynek**: *On the Complexity of Optimal Parallel Cooperative Path-Finding*. Fundamenta Informaticae, Volume 137, Number 4, pp. 517-548, IOS Press, 2015.

On the Complexity of Optimal Parallel Cooperative Path-Finding

Pavel Surynek

Charles University Prague
Faculty of Mathematics and Physics
Department of Theoretical Computer Science and Mathematical Logic
Malostranské náměstí 25, Praha, 118 00, Czech Republic
pavel.surynek@mff.cuni.cz

Abstract. A parallel version of the problem of cooperative path-finding (pCPF) is introduced in this paper. The task in CPF is to determine a spatio-temporal plan for each member of a group of agents. Each agent is given its initial location in the environment and its task is to reach the given goal location. Agents must avoid obstacles and must not collide with each other. The environment where agents are moving is modeled as an undirected graph. Agents are placed in vertices and they move along edges. At most one agent is placed in each vertex and at least one vertex remains unoccupied.

An agent can move only into a currently unoccupied vertex in the standard version of CPF. In the parallel version, an agent can move also into the vertex being currently vacated by another agent supposed this movement is not cyclic.

The optimal pCPF where the task is to find a solution of the makespan as small as possible is particularly studied. The main contribution of this paper is the proof of the *NP*-completeness of the decision version of the optimal pCPF. The reduction of propositional satisfiability (SAT) to the problem is used in the proof.

***Keywords*:** cooperative path-finding (CPF), parallelism, multi-agent, sliding puzzle, (N^2-1)-puzzle, N×N-puzzle, 15-puzzle, domain dependent planning, complexity, *NP*-completeness

1. Introduction and Motivation

This paper addresses a problem of *cooperative path-finding* (CPF) [17, 18, 22] and its parallel version. Consider a group of mobile agents that are moving in

some environment (for example in the 2-dimensional plane with obstacles). Each agent of the group is given an initial and a goal location. The question of interest is to determine a sequence of moves for each agent such that all the agents reach their goal locations supposing they started from the given initial ones by following this sequence. Physical limitations must be observed: agents must **not collide** with each other and they must **avoid obstacles**.

The CPF problem is **motivated** by many practical tasks. Various problems of navigating a group of mobile agents can be formulated as CPF. However, the primary motivations for the problem are tasks of relocating certain entities (autonomous or centrally controlled) within an environment with a limited free space. Hence, the problem is not restricted to the case where agents are actually represented by mobile agents. Such real-life examples include rearranging of stored items in automated storages (an agent is represented by a movable pile with stored items – see Figure 1) or coordination of vehicles in dense traffic (agent = vehicle). Moreover, the reasoning about rearrangement/coordination tasks should not be limited to physical entities only. An agent may be represented by a virtual entity or by a piece of commodity as well. Thus, many tasks such as planning of data transfer between communication nodes with limited storage capacity (agent = data packet), commodity transportation in the commodity transportation network (agent = certain amount of commodity), or even the motion planning of large groups of virtual agents in the computer-generated imagery can be expressed as an instance of CPF.

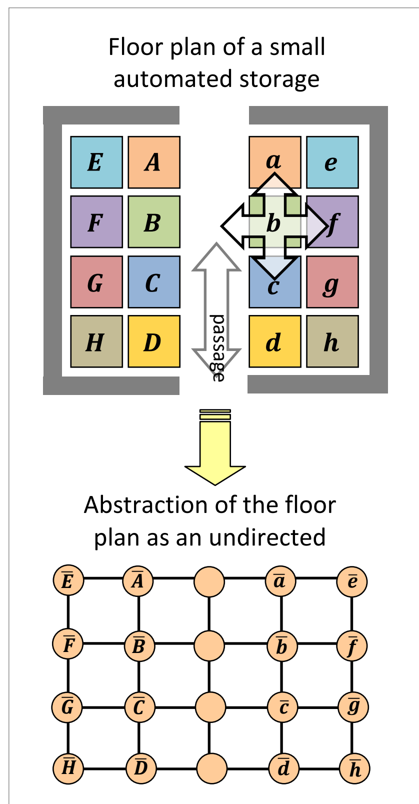


Figure 1. *Illustration of* ***modeling*** *the environment in a real scenario by undirected graph.* The scenario consists of a small automated storage with movable piles of stored items (labeled A to H and a to h). Each pile can be moved left/right/forward/backward. Items in piles are accessible from the passage – to access piles E-H or e-h the storage needs to be rearranged. The environment is modeled as grid of size 4×5 which is a bi-connected graph.

A parallel version of CPF (pCPF) is suggested in this paper and its computational complexity is studied. The standard CPF is usually formulated on an undirected graph that models the environment. Vertices of the graph represent locations and edges represent passable regions. Agents

are placed in vertices of the graph and they are allowed to move into a neighboring vertex if it is currently **unoccupied.** The parallel version of CPF is more relaxed – an agent also allowed entering a vertex that is **simultaneously vacated** by another agent supposed that agents do not perform a cyclic movement (cyclic movement includes rotation around a cycle but also swapping a pair of agents along single edge). In other words, an agent entering an unoccupied vertex, which **leads** this simultaneous movement, must exist.

An abstract instance for a given specific real-life cooperative path-finding situation can be modeled in a variety of ways. For instance, it is necessary to sample locations in the original environment in order to make the abstract instance as precise as needed. Nevertheless, these issues are out of scope of this work.

The main contribution of this paper is the proof of *NP*-completeness of the optimal pCPF. This result has been already previewed in the short conference paper [26]. However, the proof did not fit into the short paper. Here the proof is presented in all the details with rigorous treatment and illustrations.

In the context of CPF, works on problems of motion planning over graphs must be mentioned [13, 14, 15, 16, 37] since they are closely related. Namely, so-called *pebble motion on graphs* (PMG) of which the most widely known representative is the *15-puzzle* [13, 15, 16, 37] represent the standard (non-parallel) CPF in fact. Many theoretical results are known for PMG – it is known that the problem can be solved in a polynomial time (in $O(|V|^3)$ for $G = (V, E)$ modeling the environment) with solution consisting of polynomial number of moves (again $O(|V|^3)$ moves) [13, 37]. Moreover, it is known that the decision version of the optimal PMG (that is, a yes/no question if a solution of given length/makespan exists) is *NP*-complete [15, 16]. This result has been shown for a generalized variant of the 15-puzzle that is also known as $(N^2 - 1)$-puzzle. Hence, a natural question if the situation changes in the case of pCPF arises. This paper gives the answer.

The organization of the paper is as follows: the formal definition of PMG is recalled and the definition of pCPF is given in Section 2 (87). Some basic properties of both problems and their correspondence are discussed in this section as well. Section 3 (92) represents the core of the paper - several techniques for polynomial transformation of propositional satisfiability to pCPF are described here. The last section - Section 4 (116) - is devoted to related works and conclusion.

2. Pebble Motion on a Graph and Cooperative Path-finding

Problems of *pebble motion on a graph* (PMG) and *parallel cooperative path-finding* (pCPF) are formally defined in this section. As it was mentioned, non-parallel CPF and PMG are used to denote the same concept by many authors [13,

20, 37]. The PMG/CPF problem has been already studied in the literature and many theoretical results are known for this problem. The *parallel* version of CPF represents a relaxation of PMG/CPF with respect to the dynamicity of movements.

Consider an environment in which a group of mobile agents is moving. The agents are all identical (that is, they are all of the same size and have the same moving abilities). Each agent starts at a given initial location and it needs to reach a given goal location. Both problems consist in finding a spatial-temporal path for each agent so that it can reach its goal by following this path. Agents must **not collide** with each other and they must **avoid obstacles** in the environment.

An abstraction common in the literature related to PMG/CPF is adopted regarding the model of the environment [18, 20]. The environment with obstacles in which the agents are moving is modeled as an **undirected graph**. Vertices of this graph represent locations in the environment and the edges model a passable way from one location to the neighboring location. The time is discrete – each agent is located in a vertex at each time step. A motion of an agent is an instantaneous event. If the agent is placed in a vertex at a given time step then the result of the motion is the situation where the agent is placed in the neighboring vertex at the following time step.

2.1. *Formal Definitions of Motion Problems*

A notion of *pebble motion on a graph* – PMG (also called a *pebble motion puzzle*, *sliding box puzzle*; special variants are known as the *15-puzzle* and (N^2-1)-*puzzle*) [13, 16, 37] and the related problem of *cooperative path-finding* – CPF (also known as *multi-agent path-finding*) [20, 25, 32] are described in the following definition.

Definition 1 ***(pebble motion on a graph–*** **PMG).** Let $G=(V,E)$ be an undirected graph and let $P=\{\bar{p}_1,\bar{p}_2,\ldots,\bar{p}_\mu\}$ where $\mu<|V|$ be a set of *pebbles*. The *initial arrangement* and the *goal arrangement* of pebbles in G are defined by two uniquely invertible functions $S_P^0: P \longrightarrow V$ (that is $S_P^0(p) \neq S_P^0(q)$ for every $p,q \in P$ with $p \neq q$) and $S_P^+: P \longrightarrow V$ respectively. A problem of *pebble motion on a graph* (*PMG*) is the task to find a number ξ and a sequence of pebble arrangements $\mathcal{S}_P=[S_P^0,S_P^1,\ldots,S_P^\xi]$ such that the following conditions hold (the sequence represents arrangements of pebbles at each time step – the time step is indicated by the upper index):

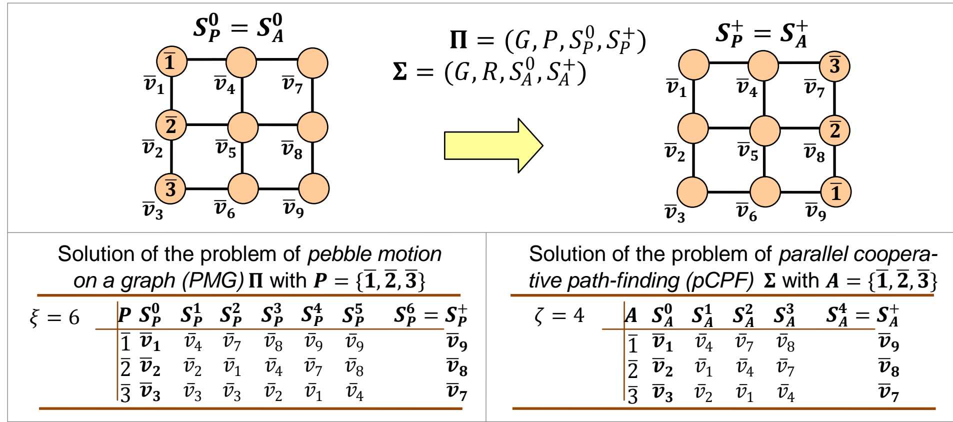
(i) $S_P^k: P \longrightarrow V$ is a uniquely invertible function for every $k=1,2,\ldots,\xi$;

(ii) $S_P^\xi = S_P^+$ (that is, all the pebbles eventually reach their destination vertices);

(iii) either $S_P^k(p) = S_P^{k+1}(p)$ or $\{S_P^k(p), S_P^{k+1}(p)\} \in E$ for every $p \in P$ and $k = 1,2,\dots,\xi - 1$ (that is, a pebble either stays in a vertex or moves along an edge);

(iv) if $S_P^k(p) \neq S_P^{k+1}(p)$ (that is, the pebble p moves between time steps k and $k+1$) then $S_P^k(q) \neq S_P^{k+1}(p)$ $\forall q \in P$ with $q \neq p$ must hold for every $p \in P$ and $k = 1,2,\dots,\xi - 1$ (that is, a pebble can move into a currently unoccupied vertex only).

The instance of PMG is formally a quadruple $\Pi = (G, P, S_P^0, S_P^+)$. A solution to the instance Π will be denoted as $\mathcal{S}_P(\Pi) = [S_P^0, S_P^1, \dots, S_P^\xi]$. □



Solution of the problem of *pebble motion on a graph (PMG)* Π with $P = \{\bar{1}, \bar{2}, \bar{3}\}$

$\xi = 6$

P	S_P^0	S_P^1	S_P^2	S_P^3	S_P^4	S_P^5	$S_P^6 = S_P^+$
$\bar{1}$	$\bar{v}_1$	$\bar{v}_4$	$\bar{v}_7$	$\bar{v}_8$	$\bar{v}_9$	$\bar{v}_9$	$\bar{v}_9$
$\bar{2}$	$\bar{v}_2$	$\bar{v}_2$	$\bar{v}_1$	$\bar{v}_4$	$\bar{v}_7$	$\bar{v}_8$	$\bar{v}_8$
$\bar{3}$	$\bar{v}_3$	$\bar{v}_3$	$\bar{v}_3$	$\bar{v}_2$	$\bar{v}_1$	$\bar{v}_4$	$\bar{v}_7$

Solution of the problem of *parallel cooperative path-finding (pCPF)* Σ with $A = \{\bar{1}, \bar{2}, \bar{3}\}$

$\zeta = 4$

A	S_A^0	S_A^1	S_A^2	S_A^3	$S_A^4 = S_A^+$
$\bar{1}$	$\bar{v}_1$	$\bar{v}_4$	$\bar{v}_7$	$\bar{v}_8$	$\bar{v}_9$
$\bar{2}$	$\bar{v}_2$	$\bar{v}_1$	$\bar{v}_4$	$\bar{v}_7$	$\bar{v}_8$
$\bar{3}$	$\bar{v}_3$	$\bar{v}_2$	$\bar{v}_1$	$\bar{v}_4$	$\bar{v}_7$

Figure 2. *An illustration of problems of* ***pebble motion on a graph (PMG)*** *and* ***parallel cooperative path-finding (pCPF)***. Both problems are illustrated on the same graph with the same initial and goal locations. The task is to move pebbles/agents from their initial locations specified by S_P^0/S_A^0 to the goal locations specified by S_P^+/S_A^+. A solution of makespan 6 ($\xi = 6$) is shown for PMG and a solution of makespan 4 ($\zeta = 4$) is shown for pCPF. Notice the differences in parallelism between both solutions – pCPF allows a higher number of moves to be performed in parallel.

The **notation** with a stripe above the symbol is used to distinguish a constant from a variable (for example, $p \in P$ is a variable while $\bar{p}_2$ is a constant; sometimes a constant parameterized by a variable or by an expression will be used – for example $\bar{p}_i$ denotes a constant parameterized by an index $i \in \mathbb{N}$; the parameterization by an expression will be clear from the context).

When speaking about a move at time step k, it is referred to the time step of commencing the move (the move is performed between time steps k and $k+1$).

A parallel version of CPF is a **relaxation** of PMG/CPF. The requirement that the target vertex of a pebble/agent must be vacated in the previous time step is relaxed. Thus, the move of an agent entering the target vertex, that is simultaneously vacated by another agent and no other agent is trying to enter the same target vertex, is allowed in parallel version of CPF. However, there must be some leading agent initiating such a chain of moves by moving into an unoccupied ver-

tex (that is, agents can move like a train with the leading agent in front) that is not entered by another agent at the same time step. These requirements rule out rotation of agents around a cycle with no vacant position as well as swapping a pair of agents along an edge. The problem is formalized in the following definition.

Definition 2 ***(parallel cooperative path-finding – pCPF).*** Again, let $G = (V, E)$ be an undirected graph. A set of *agents* $A = \{\bar{a}_1, \bar{a}_2, \dots, \bar{a}_\nu\}$ where $\nu < |V|$ is given instead of the set of pebbles. Similarly, the graph models the environment where the agents are moving. The *initial arrangement* and *the goal arrangement* of agents are defined by two uniquely invertible functions $S_A^0: A \longrightarrow V$ (that is, $S_A^0(a) \neq S_A^0(b)$ for every $a, b \in A$ with $a \neq b$) and $S_A^+: A \longrightarrow V$ respectively. A problem of *parallel cooperative path-finding* (*pCPF*) is then the task to find a number ζ and a sequence of agent arrangements $\mathcal{S}_A = [S_A^0, S_A^1, \dots, S_A^\zeta]$ for that the following conditions hold:

(i) $S_A^k: A \longrightarrow V$ is a valid arrangement for every $k = 1,2, \dots, \zeta$ (that is, uniquely invertible);

(ii) $S_A^\zeta = S_A^+$ (that is, all the agents eventually reach their destinations);

(iii) either $S_A^k(a) = S_A^{k+1}(a)$ or $\{S_A^k(a), S_A^{k+1}(a)\} \in E$ for every $a \in A$ and $k = 1,2, \dots, \zeta - 1$ (that is, an agent either stays in a vertex or moves into the neighboring vertex);

(iv) if $S_A^k(a) \neq S_A^{k+1}(a)$ (that is, the agent a moves between time steps k and $k + 1$) then there must exist a sequence of distinct agents $[a = b_0, b_1, \dots, b_\lambda]$ with $\lambda \in \mathbb{N}_0$ such that $S_A^k(c) \neq S_A^{k+1}(b_\lambda)$ $\forall c \in A$ with $c \neq b_\lambda$ (b_λ moves to a vertex that is unoccupied at time step k; b_λ is a *leading* agent of the chain of agents which the sequence is part of) and $S_A^{k+1}(b_i) = S_A^k(b_{i+1})$ for $i = 0,1, \dots, \lambda - 1$ (agents $a = b_0, b_1, \dots, b_{\lambda-1}$ follows the leader like a chain; they move all at once between time steps k and $k + 1$).

The instance of pCPF is formally a quadruple $\Sigma = (G, A, S_A^0, S_A^+)$. A solution to the instance Σ will be denoted as $\mathcal{S}_A(\Sigma) = [S_A^0, S_A^1, \dots, S_A^\zeta]$. □

The only conceptual difference between definitions of PMG/CPF and pCPF is in the point (iv). The rest of differences is attributable to different names of functions representing arrangements of agents.

The numbers ξ and ζ are called the ***makespan*** of the solution of PMG/CPF and pCPF respectively. The makespan needs to be distinguished from the ***size*** of the solution, which is the total number of moves performed by pebbles/agents. The makespan is typically less than the size of the solution. In case of the PMG/CPF with just a single unoccupied vertex, the makespan and the size of the solution are the same.

Examples of instances of PMG/CPF and pCPF and their solutions are shown in Figure 2.

2.2. *Known Properties of Motion Problems and Related Questions*

Notice that a solution of an instance of PMG/CPF as well as a solution of an instance of pCPF allows a pebble/agent to **stay** in a vertex for more than a single time step. It is also possible that a pebble/agent **visits** the same vertex **several times** within the solution. Hence, a sequence of moves for a single pebble/agent does not necessarily form a simple path in the given input graph (if the trajectory of the agent is needed to be modeled by a simple path one may need to consider a *time expanded graph* with a copy of the input graph for every time step; a path in the time expanded graph is always simple as it connects vertices in consecutive time steps only).

Notice further that both problems intrinsically allow parallel movements of pebbles/agents. That is, more than one pebble/agent can perform a move at a single time step. However, pCPF allows higher parallelism due to its weaker requirements on movements (the target vertex is required to be unoccupied only for the leading agent in the current time step – see Figure 2). More than one unoccupied vertex is necessary to obtain parallelism in PMG/CPF. On the other hand, it is sufficient to have a single unoccupied vertex to obtain parallelism in pCPF (consider for example agents moving around a cycle with one vacant position).

There is an easy to prove correspondence between solutions of PMG/CPF and pCPF summarized in the following proposition. It states that the solution of the instance of PMG/CPF can be used as a solution to the corresponding instance of the pCPF, which has the same graph, same set of agents, and the same initial and goal arrangements.

Proposition 1 ***(problem correspondence).*** Let $\Pi = (G, P, S_P^0, S_P^+)$ be an instance of PMG/CPF and let $\mathcal{S}_P(\Pi) = [S_P^0, S_P^1, \ldots, S_P^\xi]$ be its solution. Then $\mathcal{S}_A(\Sigma) = \mathcal{S}_P(\Pi)$ is a solution of an instance pCPF $\Sigma = (G, P, S_P^0, S_P^+)$ (that is, the instance of pCPF on the same graph has the set of agents represented by the set of pebbles and the initial/goal locations of agents are the same those of pebbles). ■

There is a variety of modifications of PMG/CPF and pCPF. A natural additional requirement is to produce solutions with the **shortest possible makespan** (that is, the numbers ξ or ζ respectively are required to be as small as possible). Unfortunately, this requirement makes the problem of PMG/CPF **intractable**. It was shown in [15, 16] that producing a makespan optimal solution to a special case of PMG/CPF known as $N \times N$-*puzzle* (or $(N^2 - 1)$-*puzzle*) which takes place on 4-connected grid of size $N \times N$ with one vacant position is $\boldsymbol{NP}$**-hard**.

Hence, PMG/CPF on general graph with arbitrary set of pebbles/agents is NP-hard as well.

However, it is not simply possible to make any similar statement about the complexity of the optimal pCPF based on the above facts. The situation here is complicated by the inherent parallelism, which may affect the makespan in an unforeseen way. Proof constructions used for the $N \times N$-puzzle in [15, 16] thus no longer apply for pCPF.

Observe further that difficult cases of the problem of PMG/CPF have a single unoccupied vertex. This fact may raise a question how the situation is changed when there are **more** than one **unoccupied vertices**. The intuition prompts that more unoccupied vertices may simplify the problem. Unfortunately, it is not the case. PMG/CPF on a general graph with the fixed number of unoccupied vertices is still NP-hard since multiple copies of the $N \times N$-puzzle from [15, 16] can be used to add as many unoccupied vertices as needed (notice that the resulting graph may be disconnected).

Without the requirement on the **optimality** of makespan of solutions the situation is much easier; PMG/CPF is in the ***P* class** as it is shown in [13, 37]. Due to Proposition 1 pCPF is also in the P class. It has been also shown in [13] that a solution of the size of $\mathcal{O}(|V|^3)$ can be generated for any solvable PMG/CPF instance $\Pi = (G = (V, E), P, S_P^0, S_P^+)$. Hence, it provides a polynomial upper bound on the size of the content of the oracle to guess the solution in the non-deterministic model. Thus, it is possible to conclude that decision version of optimal PMG/CPF is an ***NP*-complete** problem. By the decision version here, it is meant the yes/no question whether there is a solution of Π of the makespan smaller than the given bound.

It seems that PMG/CPF and pCPF problems have been already resolved except the case of the complexity of the optimal pCPF. However, there is another issue worth studying. Constructions proving the membership of the problem of PMG/CPF into the P class used in [13, 37] generate solutions that are too long for practical use. As the makespan of the solution is of great importance in practice, this property makes these methods unsuitable when dealing with some real life motion problem abstracted as PMG/CPF or pCPF [23, 24, 25]. Hence, alternative solving methods that generate shorter though sub-optimal solutions are also of interest [22, 23, 24, 25, 28].

3. The Intractability of the Optimal Cooperative Path-Finding

The main result is shown in this section is that the optimal pCPF is intractable. Namely, it is shown to be *NP*-hard and corresponding decision version to be *NP*-complete. The proof technique was partially inspired by Even's et al. [6] proof of

NP-completeness of *two-commodity integral flow* problem [1]. Similarly as in [6] we reduce *propositional satisfiability* (SAT) [1, 8, 12] to optimal pCPF to show its *NP*-hardness. The reduction is quite complex and requires thorough technical preparation which is elaborated in following sections. On the other hand, showing membership of optimal pCPF to *NP* is relatively easy.

3.1. *Overview of Reduction of SAT to Optimal pCPF*

As the reduction of SAT to optimal pCPF is technically complicated, a brief overview is provided now to improve readability of the technical description.

The top-level idea of the reduction is that movement of agents will simulate valuation of the given propositional formula. Thus, we need to construct an instance of pCPF for the given propositional formula so that there will be two options of going through a certain location in the graph that correspond to every propositional variable. If agents go through one of these two options the corresponding propositional variable should be valuated accordingly to either $TRUE$ or $FALSE$. The question is how to construct the pCPF instance where movements of agents in any optimal solution simulate valuation of the formula. Two fundamental properties are implicitly present when propositional formula is valuated. Both of them need to be simulated explicitly when the formula is reduced to an instance of pCPF.

The first property is so called *propositional consistency*, which means that all the positive and all the negative occurrences of the same variable in the input formula have the same propositional value respectively. The second property is the fact that all the *clauses* of the propositional formula in *CNF* [12] need to be satisfied in order to satisfy the entire formula. This characteristic will be called *clause satisfaction*.

The following section is devoted to techniques for controlling movements of agents over the graph in optimal solutions of pCPF. A so-called *vertex locking* mechanism is developed to force agents to move or not to move into some vertices of the graph. Movements of agents need to be controlled to allow the simulation of propositional consistency and clause satisfaction in pCPF eventually, which is elaborated in subsequent sections. A mechanism of so-called *conjugation* is developed to keep a group of moving agents together in order to simulate propositional consistency properly – the group must not be divided between positive and negative optional pass ways, which simulates that all the occurrences of a given variable are assigned the same truth-value. All the movement controlling techniques are finally put together to simulate finding satisfying valuation of the given propositional formula by finding makespan optimal solution to the constructed instance of pCPF.

If the reader is not interested in technical details of vertex locking, it is possible to skip directly to Section 3.3 (105).

3.2. *Vertex Locking Techniques for Controlling Movements of Agents*

A technique to **prevent** agents **from entering** a given vertex at a given set of time steps will be shown in this section. This is a crucial skill used later to force agents to move in a required way in optimal solutions in order to simulate satisfying of the propositional formula properly. To allow easier understanding of suggested concepts, the explanation follows the scheme where a simple vertex locking technique is gradually augmented to obtain eventually a technique that will be actually used in the reduction.

The vertex locking technique can be applied on an arbitrary instance of pCPF. The result of the application of the technique on the instance is that agents cannot enter a selected vertex at selected time steps in any optimal solution (the shortest possible makespan of the solution is required). The augmentation of the problem consists in adding new vertices, edges, and agents into the instance. The selection of time steps at which the vertex will not be allowed for entry by the original agents is modeled by an appropriate setting of the initial and goal locations of the newly added agents. The whole construction is formalized in the following lemma and its proof.

Lemma 1 ***(vertex locking augmentation).*** Assume the following preconditions:

(a) Let $\Sigma = (G = (V, E), A, S_A^0, S_A^+)$ be an instance of pCPF and let $v \in V$ with $S_A^0(a) \neq v \; \forall a \in A$ be a so called *locked vertex*.

(b) Next, let $T = \{t_1, t_2, \ldots, t_n\}$ where $t_i \in \mathbb{N}_0$ (*natural numbers* including 0) for $i = 1,2, \ldots, n$ and $t_1 < t_2 < \cdots < t_n$ be a set of so called *lock time steps*.

Then there exists an instance of pCPF $\Sigma' = (G' = (V', E'), A', S_{A'}^0, S_{A'}^+)$ such that $\Sigma'|_V = \Sigma$ and it **never happens** that an agent $a \in A$ enters the vertex v at any time step $t \in T$ in any **optimal** solution $\mathcal{S}_{A'}^*(\Sigma')$ (entering the vertex v at the time step t means that an agent is located in v at time step t). ■

The notation $\Sigma'|_V$ stands for a *restriction* of the pCPF problem on the set of vertices V. That is, if $\Sigma' = (G' = (V', E'), A', S_{A'}^0, S_{A'}^+)$ and $V \subseteq V'$, then $\Sigma'|_V = (G'|_V, A'|_V, S_{A'}^0|_V, S_{A'}^+|_V)$ where $G'|_V = (V, E' \cap \{\{u, v\} | u, v \in V\})$, $A'|_V = \{a \in A' | S_{A'}^0(a) \in V \wedge S_{A'}^+(a) \in V\}$, $S_{A'}^0|_V : A'|_V \longrightarrow V$ with $S_{A'}^0|_V(a) = S_{A'}^0(a) \;\; \forall a \in A'|_V$, and $S_{A'}^+|_V : A'|_V \longrightarrow V$ with $S_{A'}^+|_V(a) = S_{A'}^+(a) \; \forall a \in A'|_V$. In other words, each component of the description of the instance is naturally restricted on the smaller set of vertices. The lemma states that the augmented instance Σ' after restriction on the original set of vertices is the same as original instance Σ.

Proof. Let ζ^* be the makespan of any optimal solution of pCPF instance Σ (notice, that the number ζ^* is difficult to compute as it is shown later; but assume we know it for now).

An augmentation of the graph $G = (V, E)$ will be shown first. The set of vertices V is extended with a set of new vertices $V_X = \{\bar{u}_{t_n}, \bar{u}_{t_n-1}, \dots, \bar{u}_1, \bar{w}_1, \bar{w}_2, \dots, \bar{w}_\lambda\}$ where $\lambda = \zeta^* + t_n - t_1$. The new vertices are connected around the locked vertex v in the following way. A set of edges $E_X = \{\{\bar{u}_{t_n}, \bar{u}_{t_n-1}\}, \{\bar{u}_{t_n-1}, \bar{u}_{t_n-2}\}, \dots, \{\bar{u}_2, \bar{u}_1\}, \{\bar{u}_1, v\}, \{v, \bar{w}_1\}, \{\bar{w}_1, \bar{w}_2\}, \{\bar{w}_2, \bar{w}_3\}, \dots, \{\bar{w}_{\lambda-1}, \bar{w}_\lambda\}\}$ is added to the graph with the extended set of vertices. Thus, the augmented graph is $G' = (V' = V \cup V_X, E' = E \cup E_X)$.

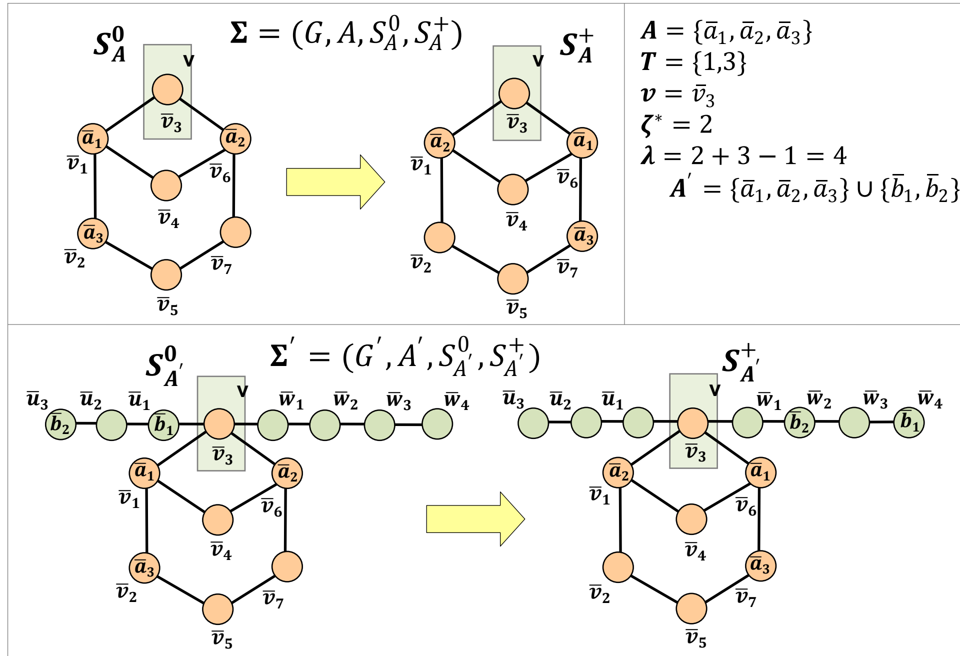


Figure 3. *An illustration of* ***vertex locking augmentation*** *in an instance of pCPF problem.* Assume we want to prevent agents $\bar{a}_1$, $\bar{a}_2$, and $\bar{a}_3$ from entering vertex $\bar{v}_3$ at time steps 1 and 3 in any optimal solution. The original instance Σ with a set of agents $A = \{\bar{a}_1, \bar{a}_2, \bar{a}_3\}$ is shown in the upper part of the figure. The makespan of any optimal solution of Σ is $\zeta^* = 2$. The augmented instance Σ' is in the lower part of the figure. New vertices $\bar{u}_3, \bar{u}_2, \bar{u}_1, \bar{w}_1, \bar{w}_2, \bar{w}_3$, and $\bar{w}_4$ and new agents $\bar{b}_1$ and $\bar{b}_2$ were added. The makespan of any optimal solution of the augmented problem is $\lambda + t_1 = \zeta^* + t_n - t_1 + t_1 = \zeta^* + t_n = 2 + 3 = 5$ and it newer happens that any of the original agents $\bar{a}_1$, $\bar{a}_2$, and $\bar{a}_3$ enters $\bar{v}_3$ at time step 1 or 3 in any optimal solution ($\bar{v}_3$ is occupied by $\bar{b}_1$ at time step 1 and by $\bar{b}_2$ at time step 3).

The idea behind the construction of the augmented graph is that new agents are initially placed in new vertices $\bar{u}_t$ for $t \in T$ with $t \geq 1$ or a new agent is placed in v if $0 \in T$. Then the newly added agents are forced to move straight

ahead into the vertices $\bar{w}_1, \bar{w}_2, \ldots, \bar{w}_\lambda$ through the vertex v. Making agents to move in this way is imposed by the condition on the optimality of the solution. Otherwise, that is if agents do not move in suggested described way, they cannot manage to reach their destinations on time. The motion of new agents through the vertex v makes an obstruction in this vertex exactly at selected time steps given by lock time steps T.

The formal description of the above idea follows. The set of agents is extended with a set of new agents $A_X = \{\bar{b}_1, \bar{b}_2, \ldots, \bar{b}_n\}$; that is, $A' = A \cup A_X$. The initial and goal arrangements of new agents are spread around the locked vertex v in the newly added vertices: $S_{A'}^0(\bar{b}_i) = \bar{u}_{t_i}$ if $t_i \neq 0$ and $S_{A'}^0(\bar{b}_i) = v$ if $t_i = 0$ for $i = 1,2, \ldots, n$; $S_{A'}^+(\bar{b}_i) = \bar{w}_{\lambda+t_1-t_i}$ if $\lambda + t_1 - t_i \geq 1$ and $S_{A'}^+(\bar{b}_i) = v$ if $\lambda + t_1 - t_i = 0$ for $i = 1,2, \ldots, n$. For the original agents, the initial and the goal arrangements remain the same; that is, $S_{A'}^0(a) = S_A^0(a)$ and $S_{A'}^+(a) = S_A^+(a)$ $\forall a \in A$.

At this point, it is necessary to show that it never happens that an agent $a \in A$ enters the vertex v at any time step $t \in T$ within the optimal solution $S_{A'}^*(\Sigma')$. Any optimal solution of the pCPF instance Σ' has the makespan of $\lambda + t_1$. Moreover, any solution $S_{A'}^*(\Sigma') = [S_{A'}^0, S_{A'}^1, \ldots, S_{A'}^{\lambda+t_1}]$ of the optimal makespan of the instance Σ', must satisfy that $S_{A'}^0(\bar{b}_i) = \bar{u}_{t_i}$, $S_{A'}^1(\bar{b}_i) = \bar{u}_{t_i-1}$, $S_{A'}^2(\bar{b}_i) = \bar{u}_{t_i-2}$, …, $S_{A'}^{t_i-1}(\bar{b}_i) = \bar{u}_1$, $S_{A'}^{t_i}(\bar{b}_i) = v$, $S_{A'}^{t_i+1}(\bar{b}_i) = \bar{w}_1$, $S_{A'}^{t_i+2}(\bar{b}_i) = \bar{w}_2$,…, $S_{A'}^{\lambda+t_1}(\bar{b}_i) = \bar{w}_{\lambda+t_1-t_i} = S_{A'}^+(\bar{b}_i)$ for $i = 1,2, \ldots, n$. This is ensured by the fact that the shortest path from $S_{A'}^0(\bar{b}_i)$ to $S_{A'}^+(\bar{b}_i)$ in G' has the length $\lambda + t_1$ and it consists of vertices $[\bar{u}_{t_i}, \bar{u}_{t_i-1}, \bar{u}_{t_i-2}, \ldots, \bar{u}_1, v, \bar{w}_1, \bar{w}_2, \ldots, \bar{w}_{\lambda+t_1-t_i}]$. Hence, **no shorter** solution in terms of the makespan exists.

However, it remains to show that the original agents from A manage to reach their destinations within the makespan of $\lambda + t_1$. This claim follows from the equality $\lambda = \zeta^* + t_n - t_1$, that is at least for ζ^* time steps the vertex v is not obstructed by any motion of newly added agents supposing they are moving straight towards their destinations. In any optimal solution of the original instance it is sufficient to enter v at most ζ^* times (notice that none of the original agents need to occupy v at the beginning). Thus, any optimal solution of the original instance can be simulated in the augmented instance while moves of the original agents are stopped at time steps when v is obstructed. Hence, the makespan of any optimal solution of Σ' is exactly $\lambda + t_1$.

It has been shown that the vertex v is obstructed at every time step $t \in T$ in any optimal solution. Hence no original agent can enter v at any time step $t \in T$. ■

The situation from Lemma 1 is illustrated in Figure 3. Notice, that it is not difficult to extend the construction from the proof of Lemma 1 on **multiple vertices** that will be **locked** at selected time steps (different sets of time steps for locking

can be used for different vertices). Another useful property of the augmented problem is summarized in the following corollary.

Corollary 1 ***(makespan preserving vertex locking).*** Assume preconditions (a) and (b) together with the following preconditions:

(c) There exists a solution $\mathcal{S}_A(\Sigma)$ of the instance $\Sigma = (G = (V, E), A, S_A^0, S_A^+)$ of the makespan ζ where $t_n \leq \zeta$.

(d) Let $v \in V$ be a locked vertex entered by an agent within $\mathcal{S}_A(\Sigma)$ at time steps $Y = \{y_1, y_2, \ldots, y_m\}$ where $y_i \in \mathbb{N}_0$ for $i = 1,2, \ldots, n$ and $y_1 < y_2 < \cdots < y_m$ and it holds that $Y \cap T = \emptyset$.

Then there exists an instance $\Sigma' = (G' = (V', E'), A', S_{A'}^0, S_{A'}^+)$ such that $\Sigma'|_V = \Sigma$ and it **never happens** that an agent $a \in A$ enters the vertex v at any time step $t \in T$ within any **optimal** solution $\mathcal{S}_{A'}^*(\Sigma')$; moreover the makespan of any optimal solution of Σ' is again ζ. ■

Proof. The construction of Σ' is almost the same as in the proof of Lemma 1 only the parameter λ is now set to $\zeta - t_1$. Then, the makespan of ζ of any optimal solution of Σ' is ensured by the construction.

The makespan is **at least** ζ since the newly added agents must go along the newly added path towards its end which cannot be carried out in smaller makespan. On the other hand, there exists a solution of the makespan ζ of the augmented instance Σ'. The vertex v needs to be occupied only at time steps $t_1, t_2, \ldots, t_n$ by the newly added agents that do not interfere with time steps at which the vertex v is entered within the solution $\mathcal{S}_A(\Sigma)$ by the original agents (this is due to $Y \cap T = \emptyset$). Altogether, the makespan of any optimal solution of augmented instance Σ' is ζ. ■

Lemma 1 as well as Corollary 1 can be generalized for locking a given number of vertices of a selected subset of vertices $W \subseteq V$ at a selected set of time steps T. Nevertheless, only a special variant of this generalization, where just one vertex of W is to be locked at selected time steps, will be actually used in further reasoning. To be more precise, at least one vertex in W is required not to be occupied by an agent from the original set of agents at any time step $t \in T$. It can be regarded as a kind of **disjunctive** locking where the set considered in disjunction is W. An analogous extension to Corollary 1 that preserves makespan additionally assumes the existence of a solution of the original instance where at least one vertex of W is unoccupied at any time step $t \in T$. These statements, which are merely a technical extension of Lemma 1 and Corollary 1, are formalized as Lemma 2 and Corollary 2.

Lemma 2 ***(set locking augmentation).*** Let the following preconditions hold:

(aa) $\Sigma = (G = (V, E), A, S_A^0, S_A^+)$ is an instance of pCPF and $W \subseteq V$ with $S_A^0(a) \notin W\ \forall a \in A$ be a so called *set of locked vertices*.

(bb) Next, let $T = \{t_1, t_2, \dots, t_n\}$ where $t_i \in \mathbb{N}_0$ (*natural numbers* including 0) for $i = 1,2,\dots,n$ and $t_1 < t_2 < \dots < t_n$ be a set of *lock time steps*.

Then there exists an instance of the problem of pCPF $\Sigma' = (G' = (V', E'), A', S_{A'}^0, S_{A'}^+)$ such that $\Sigma'|_V = \Sigma$ and it **never happens** that all the vertices of the set W are occupied by agents from the set A at any time step $t \in T$ within any **optimal** solution $\mathcal{S}_{A'}(\Sigma')$ (that is, at least one vertex from W is not occupied by an agent from A at any time step $t \in T$). ■

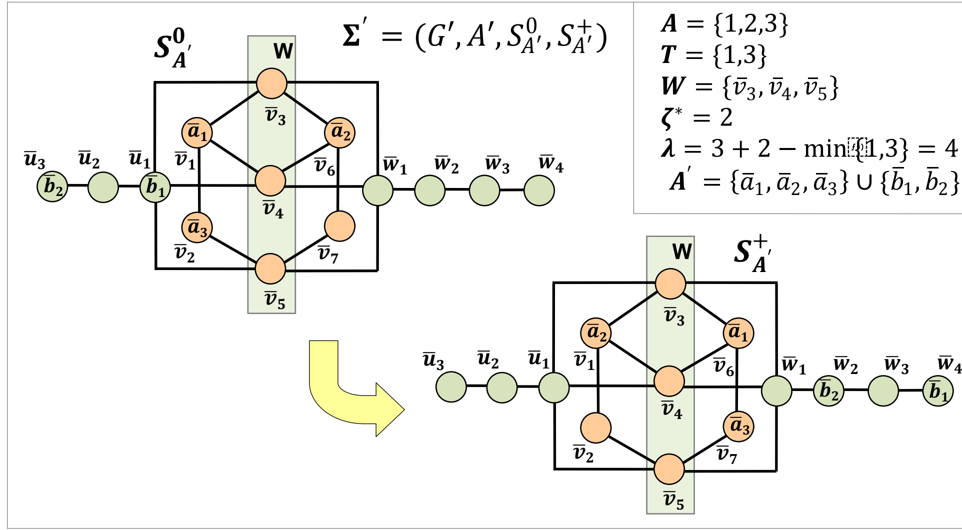


Figure 4. *An illustration of the vertex* ***set locking augmentation*** *in an instance of a pCPF problem.* Assume we want at least one vertex of the set $W = \{\bar{v}_3, \bar{v}_4, \bar{v}_5\}$ not to be occupied by any of the original agents $\bar{a}_1, \bar{a}_2$, and $\bar{a}_3$ at time steps 1 and 3 in any optimal solution. The original instance Σ with the set of agents $A = \{\bar{a}_1, \bar{a}_2, \bar{a}_3\}$ is taken from Figure 3. The augmentation is made by adding a new path consisting of vertices $\bar{u}_3, \bar{u}_2, \bar{u}_1$, $\bar{w}_1, \bar{w}_2, \bar{w}_3$, and $\bar{w}_4$ around the set W and by adding new agents $\bar{b}_1$ and $\bar{b}_2$. The makespan of any optimal solution of the augmented instance Σ' is $\lambda + 1 = t_n + \zeta^* = 3 + 2 = 5$. At least one vertex of W is occupied by $\bar{b}_1$ at time step 1 and by $\bar{b}_2$ at time step 3 in any optimal solution. Hence, it never happens that all the vertices of W are occupied by agents $\bar{a}_1, \bar{a}_2$, and $\bar{a}_3$ at time step 1 or 3 within optimal solution.

Proof. The instance Σ is augmented in a way that a new agent is forced to visit exactly one vertex of the set W at each time step $t \in T$. The technique is almost the same as in the case of Lemma 1. A path of new vertices is added around the set of locked vertices. The path branches into all the vertices of W at both connection points. Formally, the augmentation is as follows.

Let ζ^* be the makespan of any optimal solution of Σ. The set of vertices V is extended with a set of new vertices $V_X = \{\bar{u}_{t_n}, \bar{u}_{t_n - 1}, \dots, \bar{u}_1, \bar{w}_1, \bar{w}_2, \dots, \bar{w}_\lambda\}$ where

$\lambda = \zeta^* + t_n - t_1$. A set of edges $E_X = \{\{\bar{u}_{t_n}, \bar{u}_{t_n - 1}\}, \{\bar{u}_{t_n - 1}, \bar{u}_{t_n - 2}\}, ..., \{\bar{u}_2, \bar{u}_1\}, \{\bar{w}_1, \bar{w}_2\}, \{\bar{w}_2, \bar{w}_3\}, ..., \{\bar{w}_{\lambda - 1}, \bar{w}_\lambda\}\} \cup \{\{\bar{u}_1, w\} \mid w \in W\} \cup \{\{w, \bar{w}_1\} \mid w \in W\}$ is added to the graph with the extended set of vertices. Thus, the augmented graph is $G' = (V' = V \cup V_X, E' = E \cup E_X)$.

The set of agents is extended with a set of new agents $A_X = \{\bar{b}_1, \bar{b}_2, \dots, \bar{b}_n\}$; that is, $A' = A \cup A_X$. The initial and goal arrangements of new agents are spread around the set of locked vertices in the newly added vertices as follows: $S_{A'}^0(\bar{b}_i) = \bar{u}_{t_i}$ if $t_i \neq 0$ and $S_{A'}^0(\bar{b}_i) = w$ for some $w \in W$ if $t_i = 0$ for $i = 1,2, \dots, n$; $S_{A'}^+(\bar{b}_i) = \bar{w}_{\lambda + t_1 - t_i}$ if $\lambda + t_1 - t_i \geq 1$ and $S_{A'}^+(\bar{b}_i) = w$ for some $w \in W$ if $\lambda + t_1 - t_i = 0$ for $i = 1,2, \dots, n$. For the original agents, the initial and the goal arrangements remain the same; that is, $S_{A'}^0(a) = S_A^0(a)$ and $S_{A'}^+(a) = S_A^+(a)\ \forall a \in A$.

The makespan of any optimal solution of Σ' is **at least** $\lambda + t_1$ since the shortest path from $S_{A'}^0(\bar{b}_i)$ to $S_{A'}^+(\bar{b}_i)$ in G' has the length of $\lambda + t_1$ for any $i = 1,2, \dots, n$. On the other hand, since $\lambda = \zeta^* + t_n - t_1$, no vertex of W is occupied by any new agent at least for ζ^* time steps supposing the new agents are moving straight towards their destinations. Together with the fact that in any optimal solution of the original instance Σ it is sufficient to occupy W for at most ζ^* time steps, the makespan of any optimal solution of Σ' is exactly $\lambda + t_1$. ■

The construction of the augmentation from the proof of the above lemma is shown in Figure 4. Observe, that the construction can be easily extended for locking **multiple sets** of locked vertices while for each locked set different lock time steps may be used.

Corollary 2 *(makespan preserving set locking).* Assume that preconditions (aa) and (bb) hold; in addition assume that the following preconditions hold as well:

(cc) There exists a solution $\mathcal{S}_A(\Sigma)$ of the instance $\Sigma = (G = (V, E), A, S_A^0, S_A^+)$ of the makespan ζ where $t_n \leq \zeta$.

(dd) There is at least one unoccupied vertex in the selected set $W \subseteq V$ at all the time steps within $\mathcal{S}_A(\Sigma)$ except time steps $Y = \{y_1, y_2, \dots, y_m\}$ with $y_i \in \mathbb{N}_0$ for $i = 1,2, \dots, n$ and $y_1 < y_2 < \dots < y_m$ and it holds that $Y \cap T = \emptyset$.

Then there exists an instance $\Sigma' = (G' = (V', E'), A', S_{A'}^0, S_{A'}^+)$ such that $\Sigma'|_V = \Sigma$ and it **never happens** that all the vertices of W are occupied by the original agents from the set A at any time step $t \in T$ within any **optimal** solution $\mathcal{S}_{A'}^*(\Sigma')$; moreover the makespan of any optimal solution of Σ' is again ζ. ■

Proof. The construction of Σ' is almost the same as in the proof of Corollary 1. The difference is that the parameter λ is now set to $\zeta - t_1$. The construction then ensures that the makespan of any optimal solution of Σ' is ζ.

The makespan of any optimal solution is **at least** ζ since the newly added agents must go to the end of the newly added path. On the other hand, all the vertices of the set W need to be occupied by the original agents within the solution $\mathcal{S}_A(\Sigma)$ only at time steps $y_1, y_2, \dots, y_m$ that do not interfere with time steps $t_1, t_2, \dots, t_n$ (since $Y \cap T = \emptyset$) at which the newly added agents need to occupy at least one vertex of W (supposing they are going directly to their destinations along the newly added path). Hence, there exists a solution of the makespan ζ of the augmented instance Σ'. Altogether, any optimal solution of Σ' has the makespan ζ. ■

Observe that original agents are allowed to enter newly added vertices in all the above augmentations. This may help the original agents to reach their destinations **faster** (the newly added vertices may be used as additional "parking place" for agents). This behavior of agents is undesirable in the planned reduction where it is needed to isolate vertex locking mechanism from the original instance. Hence, a slight adaptation of the vertex locking technique must be used.

Some additional notations are needed to express the requirement on not using the newly added vertices by the original agents formally. Let $\mathcal{S}^*_{A'}(\Sigma') = [\mathcal{S}^0_{A'}, \mathcal{S}^1_{A'}, \dots, \mathcal{S}^\zeta_{A'}]$ be an optimal solution of the pCPF instance Σ' over the graph $G' = (V', E')$ and let $V \subseteq V'$. Then the restriction of the solution $\mathcal{S}^*_{A'}(\Sigma')$ on the set of vertices V is denoted as $\mathcal{S}^*_{A'}(\Sigma')|_V = [\mathcal{S}^0_{A'}|_V, \mathcal{S}^1_{A'}|_V, \dots, \mathcal{S}^\zeta_{A'}|_V]$, where $\mathcal{S}^i_{A'}|_V : A'|_V \longrightarrow V$ with $\mathcal{S}^i_{A'}|_V(a) = \mathcal{S}^i_{A'}(a)$ $\forall a \in A'|_V$ for $i = 0,1,\dots,\zeta$. Next, let $Sol^*(\Sigma') = \{\mathcal{S}^*_{A'}(\Sigma') | \mathcal{S}^*_{A'}(\Sigma') \text{ is an optimal solution of } \Sigma'\}$, then $Sol^*(\Sigma')|_V = \{\mathcal{S}^*_{A'}(\Sigma')|_V | \ \mathcal{S}^*_{A'}(\Sigma') \in Sol^*(\Sigma')\}$, and let $Sol(\Sigma) = \{\mathcal{S}_A(\Sigma) | \ \mathcal{S}_A(\Sigma) \text{ is a solution}$ (not necessarily optimal) of $\Sigma\}$. An augmentation Σ' of the instance Σ where added vertices are never used can be expressed by the condition $Sol^*(\Sigma')|_V \subseteq Sol(\Sigma)$.

Proposition 2 ***(two-stage vertex locking).*** Assume preconditions (a) and (b). Then there exists an instance of pCPF $\Sigma^\sim = (G^\sim = (V^\sim, E^\sim), A^\sim, \mathcal{S}^0_{A^\sim}, \mathcal{S}^+_{A^\sim})$ such that $\Sigma^\sim|_V = \Sigma$ where it **never happens** that an agent $a \in A$ enters v at any time step $t \in T$ within any **optimal** solution $\mathcal{S}_{A^\sim}(\Sigma^\sim)$ and $Sol^*(\Sigma^\sim)|_V \subseteq Sol(\Sigma)$ (that is, **original** agents cannot use any added vertex in any optimal solution). ■

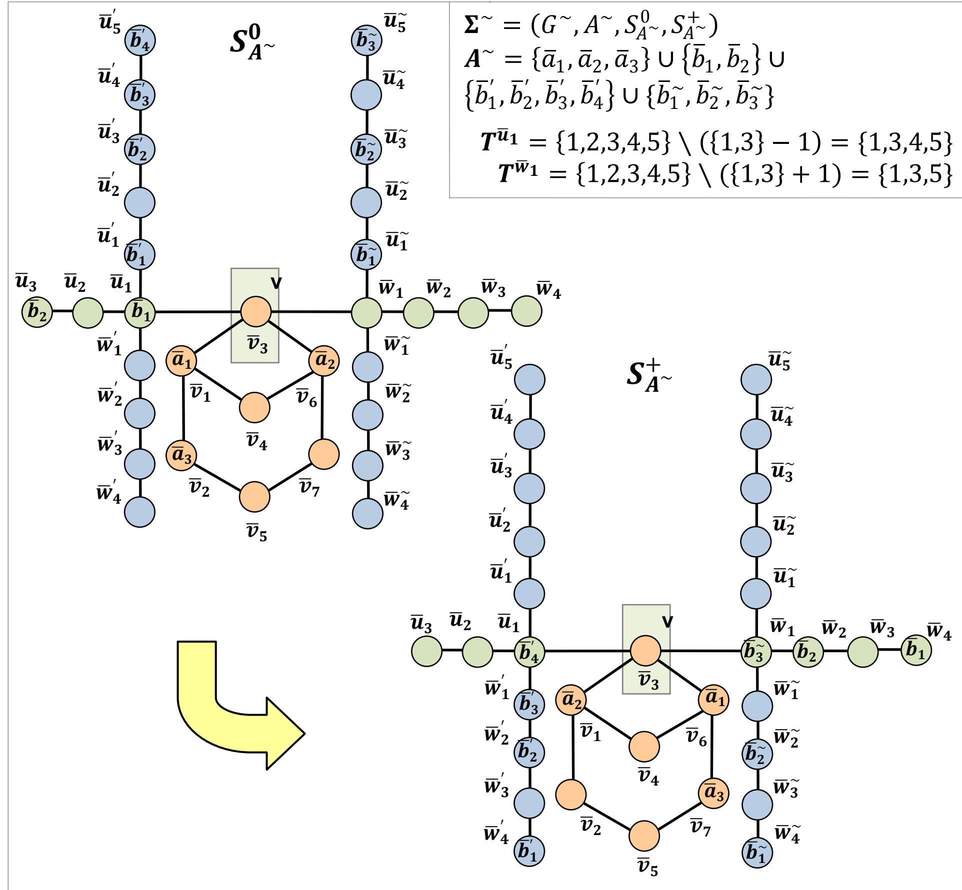


Figure 5. *An illustration of* ***two-stage vertex locking*** *in an instance of the pCPF problem.* Assume we want agents $\bar{a}_1$, $\bar{a}_2$, and $\bar{a}_3$ prevent from entering vertex $\bar{v}_3$ at time steps 1 and 3. Additionally no vertex added by the augmentation can be entered by the original agents $\bar{a}_1$, $\bar{a}_2$, and $\bar{a}_3$. These requirements are ensured by two stage locking. First, $\bar{v}_3$ is locked at time steps 1 and 3 using a path of new vertices $\bar{u}_3$, $\bar{u}_2$, $\bar{u}_1$, $\bar{w}_1$, $\bar{w}_2$, $\bar{w}_3$, and $\bar{w}_4$ (this stage corresponds to Figure 3). Then $\bar{u}_1$ and $\bar{w}_1$ are locked at time steps $T^{\bar{u}_1} = \{1,3,4,5\}$ and $T^{\bar{w}_1} = \{1,3,5\}$ respectively by the same technique. The makespan of any optimal solution of $\Sigma^\sim$ is 5 (the same as of Σ').

Notice, that Proposition 2 (100) is almost the same as Lemma 1 except the additionally required condition $Sol^*(\Sigma^\sim)|_V \subseteq Sol(\Sigma)$.

Proof. The basic construction from the proof of Lemma 1 will be adopted; then some further augmentations will be made by successive applications of Corollary 1 to enforce the condition that $Sol^*(\Sigma^\sim)|_V \subseteq Sol(\Sigma)$.

Let ζ^* denote be the makespan of optimal solutions of Σ. In the **first stage**, the graph G is extended exactly as in the previous case. That is, the set of vertices $V_X = \{\bar{u}_{t_n}, \bar{u}_{t_n-1}, \dots, \bar{u}_1, \bar{w}_1, \bar{w}_2, \dots, \bar{w}_\lambda\}$ where $\lambda = \zeta^* + t_n - t_1$ and the set of edges $E_X = \{\{\bar{u}_{t_n}, \bar{u}_{t_n-1}\}, \{\bar{u}_{t_n-1}, \bar{u}_{t_n-2}\}, \dots, \{\bar{u}_2, \bar{u}_1\}, \{\bar{u}_1, v\}, \{v, \bar{w}_1\}, \{\bar{w}_1, \bar{w}_2\},$

$\{\bar{w}_2, \bar{w}_3\}$, ..., $\{\bar{w}_{\lambda-1}, \bar{w}_\lambda\}\}$ are added to the graph; that is $G^{'} = (V^{'} = V \cup V_X, E^{'} = E \cup E_X)$. The set of agents is extended with $A_X = \{\bar{b}_1, \bar{b}_2, \ldots, \bar{b}_n\}$; that is, $A^{'} = A \cup A_X$ and the initial and goal arrangements of new agents are set as follows: $S_{A'}^0(\bar{b}_i) = \bar{u}_{t_i}$ if $t_i \neq 0$ and $S_{A'}^0(\bar{b}_i) = v$ if $t_i = 0$ for $i = 1,2,\ldots,n$; $S_{A'}^+(\bar{b}_i) = \bar{w}_{\lambda+t_1-t_i}$ if $\lambda + t_1 - t_i \geq 1$ and $S_{A'}^+(\bar{b}_i) = v$ if $\lambda + t_1 - t_i = 0$ for $i = 1,2,\ldots,n$. As it has been shown, this construction suffices for satisfying almost all the requirements except $Sol^*(\Sigma^{'})|_V \subseteq Sol(\Sigma)$.

Now, it is necessary to prevent agents of A from entering any of the added vertices V_X. Observe that it is sufficient to lock vertices $\bar{u}_1$ and $\bar{w}_1$ to fulfill this requirement since the newly added vertices form a path around v and this is the only vertex through which the path is connected to the original graph (neighboring vertices of v are $\bar{u}_1$ and $\bar{w}_1$). Vertices $\bar{u}_1$ and $\bar{w}_1$ need to be locked for all the time steps except time steps at which agents from the set A_X go through them in an optimal solution – this will be the **second stage** locking. More precisely, the vertex $\bar{u}_1$ needs to be locked at time steps from the set $T^{\bar{u}_1} = \{1,2,\ldots,\lambda + t_1\} \setminus (\{t_1, t_2, \ldots, t_n\} - 1)$ (where $\{t_1, t_2, \ldots, t_n\} - 1 = \{t_1 - 1, t_2 - 1, \ldots, t_n - 1\}$) and the vertex $\bar{w}_1$ needs to be locked at time steps from the set $T^{\bar{w}_1} = \{1,2,\ldots,\lambda + t_1\} \setminus (\{t_1, t_2, \ldots, t_n\} + 1)$. Notice, that $\max(T^{\bar{u}_1}) \leq \lambda + t_1$ as well as $\max(T^{\bar{w}_1}) \leq \lambda + t_1$. Moreover, the construction of sets $T^{\bar{u}_1}$ and $T^{\bar{w}_1}$ ensures that vertices $\bar{u}_1$ and $\bar{w}_1$ respectively will be locked at time steps at which they are not entered within some solution (which is known to be an optimal solution). Hence, Corollary 1 applies for $\Sigma^{'}$, the locked vertex $\bar{u}_1$, and the set of lock time steps $T^{\bar{u}_1}$; that is, optimal makespan is preserved. In other words, the vertex locking is **synchronized** with the vertex locking from the first stage. Then Corollary 1 is applied once more for the resulting instance, the locked vertex $\bar{w}_1$, and the set of lock time steps $T^{\bar{w}_1}$. Let $\Sigma^{\sim} = (G^{\sim} = (V^{\sim}, E^{\sim}), A^{\sim}, S_{A^{\sim}}^0, S_{A^{\sim}}^+)$ denote the final instance, then $Sol^*(\Sigma^{\sim})|_V \subseteq Sol(\Sigma)$. ■

The construction from Proposition 2 is illustrated in Figure 5. It is a further augmentation of the instance from Figure 3 in fact.

The important property is that the **size** of all the augmented instances of the problem is $\mathcal{O}(\max(\zeta^*, t_n) + |V| + |E|)$ where ζ^* is the optimal makespan (that is, asymptotically as many as $\max(\zeta^*, t_n)$ vertices and agents are added). Consequently, if an augmented instance is needed to be kept small (with respect to $|V| + |E|$), the numbers ζ^* and t_n must be small as well.

Corollary 3 ***(makespan preserving two-stage vertex locking).*** Assume preconditions (a), (b), (c), and (d). Then there exists an instance $\Sigma^{\sim} = (G^{\sim} = (V^{\sim}, E^{\sim}), A^{\sim}, S_{A^{\sim}}^0, S_{A^{\sim}}^+)$ such that $\Sigma^{\sim}|_V = \Sigma$ and it **never happens** that an agent $a \in A$ enters the locked vertex v at any time step $t \in T$ within any **optimal** solution $\mathcal{S}_{A^{\sim}}^*(\Sigma^{\sim})$ and $Sol^*(\Sigma^{\sim})|_V \subseteq Sol(\Sigma)$ (that is, **original** agents cannot use any

added vertex in any optimal solution); moreover the makespan of any optimal solution of $\Sigma^{\sim}$ is again ζ. ■

Proof. The construction from the proof of Proposition 2 can be adopted with a minor change. In the first stage of the construction of $\Sigma^{\sim}$ where the construction from the proof of Lemma 1 has been applied, Corollary 1 is applied instead. This ensures that the intermediate instance after the first stage locking preserves the makespan of ζ. The rest of the proof can be applied without any change. ■

Again it is not difficult to generalize the construction for locking a subset of certain size of a selected set of vertices at given time steps where the original agents can move only in the original vertices. These merely technical extensions of Proposition 2 and Corollary 3 are listed as Proposition 3 and Corollary 4.

Proposition 3 *(two-stage set locking)*. Assume that preconditions (aa) and (bb) hold. Then there exists an instance of the problem of pCPF $\Sigma^{\sim} = (G^{\sim} = (V^{\sim}, E^{\sim}), A^{\sim}, S_{A^{\sim}}^{0}, S_{A^{\sim}}^{+})$ such that $\Sigma^{\sim}|_V = \Sigma$ where it **never happens** that all the vertices of W are occupied by the original agents from A at any time step $t \in T$ within any **optimal** solution $S_{A^{\sim}}^{*}(\Sigma^{\sim})$ and $Sol^{*}(\Sigma^{\sim})|_V \subseteq Sol^{*}(\Sigma)$ (that is, **original** agents cannot use any added vertex in any optimal solution). ■

Proof. The proof will partially adopt the basic idea of the construction from the proof of Proposition 2. The vertex set locking will be done in two stages by a successive applications of Corollary 1 to enforce the condition $Sol^{*}(\Sigma^{\sim})|_V \subseteq Sol^{*}(\Sigma)$.

Let ζ^{*} be the makespan of optimal solutions of the pCPF instance Σ. The first stage of the augmentation will be done as in the case of Proposition 2. A set of vertices $V_X = \{\bar{u}_{t_n}, \bar{u}_{t_n-1}, \dots, \bar{u}_1, \bar{w}_1, \bar{w}_2, \dots, \bar{w}_\lambda\}$ where $\lambda = \zeta^{*} + t_n - t_1$ and a set of edges $E_X = \{\{\bar{u}_{t_n}, \bar{u}_{t_n-1}\}, \{\bar{u}_{t_n-1}, \bar{u}_{t_n-2}\}, \dots, \{\bar{u}_2, \bar{u}_1\}, \{\bar{w}_1, \bar{w}_2\}, \{\bar{w}_2, \bar{w}_3\}, \dots,$ $\{\bar{w}_{\lambda-1}, \bar{w}_\lambda\}\} \cup \{\{\bar{u}_1, w\} \mid w \in W\} \cup \{\{w, \bar{w}_1\} \mid w \in W\}$ are added to the graph; that is $G' = (V' = V \cup V_X, E' = E \cup E_X)$. The set of agents is extended with a set of new agents $A_X = \{\bar{b}_1, \bar{b}_2, \dots, \bar{b}_n\}$; that is, $A' = A \cup A_X$ and the initial and goal arrangements of new agents are set as follows: $S_{A'}^{0}(\bar{b}_i) = \bar{u}_{t_i}$ if $t_i \neq 0$ and $S_{A'}^{0}(\bar{b}_i) = w$ for some $w \in W$ if $t_i = 0$ for $i = 1,2,\dots,n$; $S_{A'}^{+}(\bar{b}_i) = \bar{w}_{\lambda+t_1-t_i}$ if $\lambda + t_1 - t_i \geq 1$ and $S_{A'}^{+}(\bar{b}_i) = w$ for some $w \in W$ if $\lambda + t_1 - t_i = 0$ for $i = 1,2,\dots,n$.

To prevent agents of A from entering any of the added vertices V_X **second stage** vertex locking must be done. It is sufficient to **lock** vertices u_1 and w_1 since these two vertices are the only connection points of the original graph with the newly added parts. Vertices u_1 and w_1 need to be locked for all the time steps except time steps at which agents of A_X go through them in an optimal solution.

More precisely, the vertex $\bar{u}_1$ needs to be locked for time steps from the set $T^{\bar{u}_1} = \{1,2,\dots,\lambda + t_1\} \setminus (\{t_1, t_2, \dots, t_n\} - 1)$ and the vertex $\bar{w}_1$ needs to be locked for time steps from the set $T^{\bar{w}_1} = \{1,2,\dots,\lambda + t_1\} \setminus (\{t_1, t_2, \dots, t_n\} + 1)$.

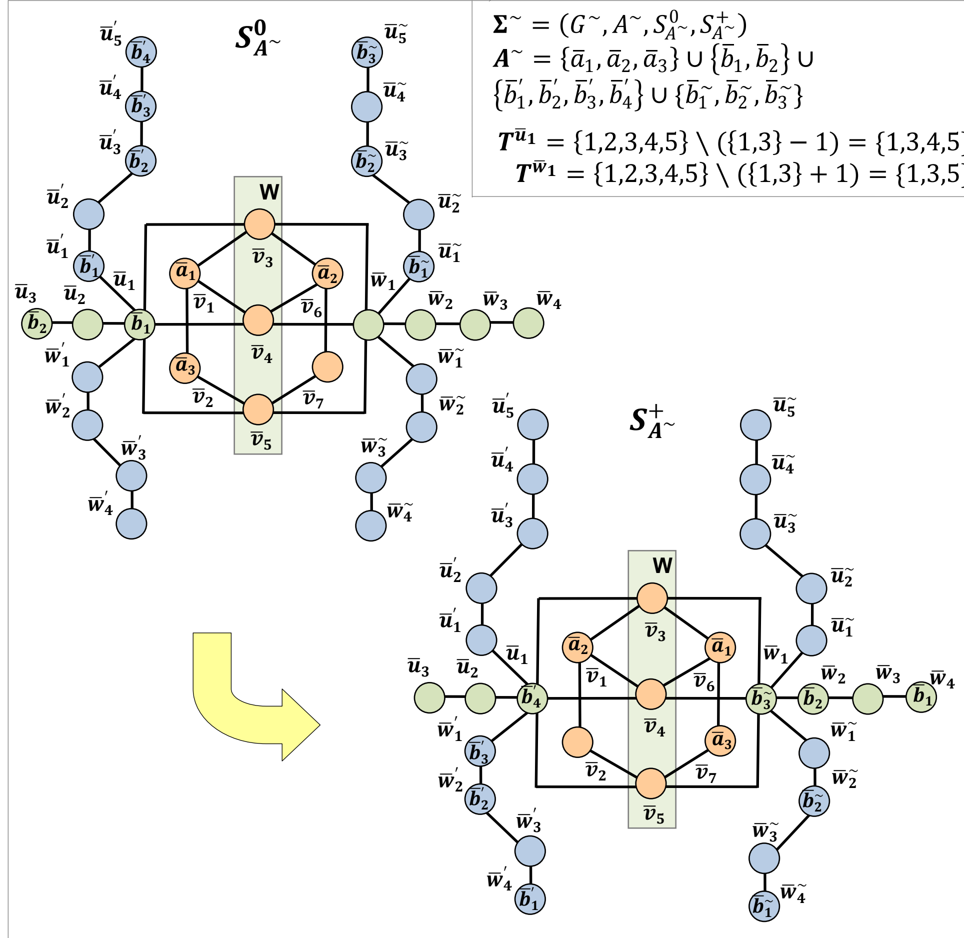


Figure 6. *An illustration of two-stage vertex set locking in an instance of the pCPF problem.* At least one vertex of the set $W = \{\bar{v}_3, \bar{v}_4, \bar{v}_5\}$ must not be occupied by any of the original agents $\bar{a}_1$, $\bar{a}_2$, and $\bar{a}_3$ at time steps 1 and 3. Additionally, no vertex added by the augmentation can be entered by any of original agents. These requirements are ensured by two stage set locking. First, the set W is locked at time steps 1 and 3 by adding a path of new vertices $\bar{u}_3$, $\bar{u}_2$, $\bar{u}_1$, $\bar{w}_1$, $\bar{w}_2$, $\bar{w}_3$, and $\bar{w}_4$ (this stage corresponds to Figure 4). Then $\bar{u}_1$ and $\bar{w}_1$ are locked at time steps $T^{\bar{u}_1} = \{1,3,4,5\}$ and $T^{\bar{w}_1} = \{1,3,5\}$ respectively by vertex locking technique. The makespan of any optimal solution of $\Sigma^\sim$ is 5 (the same as of $\Sigma^{'}$ from Figure 4).

Since $\max(T^{\bar{u}_1}) \leq \lambda + t_1$ (as well as $\max(T^{\bar{w}_1}) \leq \lambda + t_1$) and vertex $\bar{u}_1$ is to be locked for time steps at which it is not entered within some optimal solution, Corollary 1 applies for $\Sigma^{'}$, the locked vertex $\bar{u}_1$, and the set of lock time steps $T^{\bar{u}_1}$. That is, optimal makespan is preserved. Again, the vertex locking is **syn-**

chronized with the vertex locking from the first stage. Then Corollary 1 is applied once more on the resulting instance with the locked vertex $\overline{w}_1$ and the set of lock time steps $T^{\overline{w}_1}$. Let $\Sigma^{\sim} = (G^{\sim} = (V^{\sim}, E^{\sim}), A^{\sim}, S_{A^{\sim}}^{0}, S_{A^{\sim}}^{+})$ denote the final instance, then $Sol^{*}(\Sigma^{\sim})|_{V} \subseteq Sol(\Sigma)$. ■

The construction of the two-stage vertex locking from the above proof is shown in Figure 6. As in the case of locking a single vertex, the **size** of all the augmented instances of the problem is $O(\max(\zeta^{*}, t_n) + |V| + |E|)$ where ζ^{*} is the optimal makespan of Σ.

Corollary 4 ***(makespan preserving two-stage set locking).*** Assume that preconditions (aa), (bb), (cc), and (dd) hold. Then there exists an instance $\Sigma^{\sim} = (G^{\sim} = (V^{\sim}, E^{\sim}), A^{\sim}, S_{A^{\sim}}^{0}, S_{A^{\sim}}^{+})$ such that $\Sigma^{\sim}|_{V} = \Sigma$ and it **never happens** that all the vertices of W are occupied by the original agents of A at any time step $t \in T$ within any **optimal** solution $S_{A^{\sim}}^{*}(\Sigma^{\sim})$; moreover the makespan of any optimal solution of $\Sigma^{\sim}$ is again ζ and and $Sol^{*}(\Sigma^{\sim})|_{V} \subseteq Sol(\Sigma)$ (that is, **original** agents cannot use any added vertex in any optimal solution). ■

Proof. The construction of $\Sigma^{\sim}$ from the proof of Proposition 3 can be adopted with a minor change. Instead of using the construction from the proof of Lemma 1 in the first stage, Corollary 1 is applied instead. This ensures that the intermediate instance after the first stage locking preserves the makespan of ζ. The rest of the proof can be applied without any change. ■

3.3. *Conjugation - Moving Agents Together to Simulate Propositional Consistency*

We will simulate valuation of variables of the propositional formula by passing certain pass ways in the graph. There will be two pass ways for each variable – one representing positive valuation and the other negative valuation. Since we need to preserve **propositional consistency** (positive and negative literals of the same propositional variable should have **complementary** values) a group of agents for valuating a given variable must not split between these two pass ways. All the agents must pass either the positive branch or the negative branch. Hence, we need some technique that keeps a group of agents together even though they can choose between two alternative pass ways. A technique that ensures such a behavior of agents will be called a ***conjugation technique***.

It is possible to look aside from the eventual application of the conjugation technique in simulating valuation of the propositional formula. Let us now concentrate just on preventing a group of agents from splitting.

Let $A = \{\bar{c}_1, \bar{c}_2, \dots, \bar{c}_n\}$ be a set of agents that are to be conjugated. Formally, the **conjugation** means that there is an instance of the problem of path-finding for multiple agents $\Xi = (G = (V, E), A', S^0_{A'}, S^+_{A'})$, where $V = V^0 \cup V^{\mathcal{L}} \cup V^{\mathcal{R}} \cup V^+$; $V^0, V^{\mathcal{L}}, V^{\mathcal{R}}$, V^+ are pair-wise disjoint, $|V^{\mathcal{L}}| = |V^{\mathcal{R}}| = |A|$, $A \subseteq A'$, $S^0_{A'}(A) \subseteq V^0$ (image of the set by $S^0_{A'}$ is defined naturally: $S^0_{A'}(A) = \{v | (\exists a \in A) S^0_{A'}(a) = v\}$), $S^+_{A'}(A) \subseteq V^+$, and there **exists** a time step t such that within any optimal solution $\mathcal{S}^*_{A'}(\Xi) = [S^0_{A'}, S^1_{A'}, \dots, S^{\zeta}_{A'}]$ either $\{S^t_{A'}(\bar{c}_1), S^t_{A'}(\bar{c}_2), \dots, S^t_{A'}(\bar{c}_n)\} \subseteq V^{\mathcal{L}}$ or $\{S^t_{A'}(\bar{c}_1), S^t_{A'}(\bar{c}_2), \dots, S^t_{A'}(\bar{c}_n)\} \subseteq V^{\mathcal{R}}$ holds. That is, at time step t the whole group of agents that are conjugated appears either in $V^{\mathcal{L}}$ or $V^{\mathcal{R}}$. All the other cases, in which some of conjugated agents appear in $V^{\mathcal{L}}$ and some in $V^{\mathcal{R}}$, cannot appear in any optimal solution.

To rule out trivial cases of Ξ a requirement that $\{\mathcal{S}^*_{A'}(\Xi) | \; \mathcal{S}^*_{A'}(\Xi) = [S^0_{A'}, \dots, S^{\zeta}_{A'}] \wedge \{S^t_{A'}(\bar{c}_1), \dots, \; S^t_{A'}(\bar{c}_n)\} \subseteq V^{\mathcal{L}}\} \neq \emptyset$ and $\{\mathcal{S}^*_{A'}(\Xi) | \mathcal{S}^*_{A'}(\Xi) = [S^0_{A'}, \dots, S^{\zeta}_{A'}] \wedge \{S^t_{A'}(\bar{c}_1), \dots, S^t_{A'}(\bar{c}_n)\} \subseteq V^{\mathcal{R}}\} \neq \emptyset$ should be taken into account. That is, agents A must go through one of the two alternative pass ways represented by $V^{\mathcal{L}}$ and $V^{\mathcal{R}}$. The task is now to build such an instance of the pCPF problem.

The main idea of the construction is to order the agents of A into the queue that starts with an additional agent called a ***leading agent***. There is a branching in the graph into $V^{\mathcal{L}}$ and $V^{\mathcal{R}}$, which are then joined together, and two leading agents prepared. The destination for leading agents is temporarily closed by the construction from Corollary 1. This prevents the leading agents from **escaping** before fulfilling their task. The destination for agents of A is accessible from both branches of $V^{\mathcal{L}}$ and $V^{\mathcal{R}}$ **symmetrically**. The leading agents have no other choice than to lead the group of agents to their destinations. Finally, the leading agent has to go out of the way.

The crucial observation is that if the group of agents A is split between both branches, then the leading agents inevitably block each other and obstruction occurs after which there is no chance to reach destinations on time (that is, the solution cannot be finished to be optimal). Hence, agent must go into one of the branches of $V^{\mathcal{L}}$ or $V^{\mathcal{R}}$ together (they must conjugate). Below is the formal description of the construction.

The graph $G = (V, E)$ consists of the following sets of **vertices**:

$V^0 = \{\bar{v}^0_1, \bar{v}^0_2, \dots, \bar{v}^0_n\}$
(called ***initial*** *vertices*),

$V^{\mathcal{L}} = \{\bar{v}^{\mathcal{L}}_1, \bar{v}^{\mathcal{L}}_2, \dots, \bar{v}^{\mathcal{L}}_n\}$
(called ***left*** *vertices*),

$V^{\mathcal{R}} = \{\bar{v}^{\mathcal{R}}_n, \bar{v}^{\mathcal{R}}_{n-1}, \dots, \bar{v}^{\mathcal{R}}_1\}$
(called ***right*** *vertices*),

$V^+ = V^+_{\mathcal{L}} \cup V^+_{\mathcal{R}} \cup V^+_G \cup V^+_Y$
(called ***destination*** *vertices*), with

$V_{\mathcal{L}}^{+} = \{\bar{v}_{+}^{\mathcal{L}}, \bar{v}_{-2}^{\mathcal{L}}, \bar{v}_{-1}^{\mathcal{L}}, \bar{v}_{0}^{\mathcal{L}}\}$
(called ***left part*** of destination vertices)
$V_{\mathcal{R}}^{+} = \{\bar{v}_{+}^{\mathcal{R}}, \bar{v}_{0}^{\mathcal{R}}, \bar{v}_{-1}^{\mathcal{R}}, \bar{v}_{-2}^{\mathcal{R}}\}$
(called ***right part*** of destination vertices)
$V_{G}^{+} = \{\bar{v}_{1}^{+}, \bar{v}_{2}^{+}, \dots, \bar{v}_{n}^{+}\}$
(called ***gate part*** of destination vertices) and
$V_{Y}^{+} = \{\bar{v}_{1,1}^{+}, \bar{v}_{1,2}^{+}, \dots, \bar{v}_{1,n}^{+}, \bar{v}_{2,1}^{+}, \bar{v}_{2,2}^{+}, \dots, \bar{v}_{2,n}^{+}, \dots, \dots, \bar{v}_{\vartheta,1}^{+}, \bar{v}_{\vartheta,2}^{+}, \dots, \bar{v}_{\vartheta,n}^{+}\}$
(called ***array part*** of destination vertices),

where ϑ is a parameter determining the length of a solution; it is required that $\vartheta \geq n + 4$. Notice that V_{Y}^{+} is in fact an array of ϑ rows of n vertices within V^{+}. In total, the set of vertices is $V = V^{0} \cup V^{\mathcal{L}} \cup V^{\mathcal{R}} \cup V^{+}$.

The **edges** of the graph are as follows:

$E^{0} = \{\{\bar{v}_{1}^{0}, \bar{v}_{1}^{\mathcal{L}}\}, \{\bar{v}_{2}^{0}, \bar{v}_{2}^{\mathcal{L}}\}, \dots, \{\bar{v}_{n}^{0}, \bar{v}_{n}^{\mathcal{L}}\}\} \cup \{\{\bar{v}_{1}^{0}, \bar{v}_{n}^{\mathcal{R}}\}, \{\bar{v}_{2}^{0}, \bar{v}_{n-1}^{\mathcal{R}}\}, \dots, \{\bar{v}_{n}^{0}, \bar{v}_{1}^{\mathcal{R}}\}\}$
(edges for making a connection between **initial** vertices and **left/right** vertices),

$E^{-} = \{\{\bar{v}_{-2}^{\mathcal{L}}, \bar{v}_{-1}^{\mathcal{L}}\}, \{\bar{v}_{-1}^{\mathcal{L}}, \bar{v}_{0}^{\mathcal{L}}\}, \{\bar{v}_{0}^{\mathcal{L}}, \bar{v}_{1}^{\mathcal{L}}\}\} \cup \{\{\bar{v}_{-2}^{\mathcal{R}}, \bar{v}_{-1}^{\mathcal{R}}\}, \{\bar{v}_{-1}^{\mathcal{R}}, \bar{v}_{0}^{\mathcal{R}}\}, \{\bar{v}_{0}^{\mathcal{R}}, \bar{v}_{1}^{\mathcal{R}}\}\}$
(edges for connecting the remaining **left/right** vertices),

$E^{+} = \{\{\bar{v}_{0}^{\mathcal{L}}, \bar{v}_{\mathcal{L}}^{+}\}, \{\bar{v}_{L}^{+}, \bar{v}_{1}^{+}\}, \{\bar{v}_{1}^{+}, \bar{v}_{2}^{+}\}, \{\bar{v}_{2}^{+}, \bar{v}_{3}^{+}\}, \dots, \{\bar{v}_{n-1}^{+}, \bar{v}_{n}^{+}\}, \{\bar{v}_{n}^{+}, \bar{v}_{\mathcal{R}}^{+}\}, \{\bar{v}_{\mathcal{R}}^{+}, \bar{v}_{0}^{\mathcal{R}}\}\}$
(edges for connecting **left/right** vertices to the **gate part** of destination vertices),

$E_{G}^{+} = \{\{\bar{v}_{1}^{+}, \bar{v}_{1,1}^{+}\}, \{\bar{v}_{1}^{+}, \bar{v}_{1,2}^{+}\}, \dots, \{\bar{v}_{1}^{+}, \bar{v}_{1,n}^{+}\}\}$
(edges for connecting the **gate part** to the array part of **destination** vertices),

$E_{Y}^{+} = \{\{\bar{v}_{1,1}^{+}, \bar{v}_{2,1}^{+}\}, \dots, \{\bar{v}_{1,n}^{+}, \bar{v}_{2,n}^{+}\}, \dots, \dots, \{\bar{v}_{\vartheta-1,1}^{+}, \bar{v}_{\vartheta,1}^{+}\}, \dots, \{\bar{v}_{\vartheta-1,n}^{+}, \bar{v}_{\vartheta,n}^{+}\}\}$
(edges for connecting rows of the **array part** of destination vertices),

$E_{\times}^{+} = \{\{\bar{v}_{\vartheta-1,1}^{+}, \bar{v}_{\vartheta,n}^{+}\}, \{\bar{v}_{\vartheta-1,2}^{+}, \bar{v}_{\vartheta,n-1}^{+}\}, \dots, \{\bar{v}_{\vartheta-1,n}^{+}, \bar{v}_{\vartheta,1}^{+}\}\}$
(edges for connecting the **last row** of the **array part** in the **reversed** order);

in total, the set of edges of the graph G is $E = E^{0} \cup E^{-} \cup E^{+} \cup E_{G}^{+} \cup E_{Y}^{+} \cup E_{\times}^{+}$.

The **set of agents** is extended with two leading agents $\bar{l}_{\mathcal{L}}$ and $\bar{l}_{\mathcal{R}}$ (the left and the right leading agent); that is, $A^{'} = A \cup \{\bar{l}_{\mathcal{L}}, \bar{l}_{\mathcal{R}}\}$. The **initial arrangement** of agents is as follows: $S_{A'}^{0}(\bar{c}_i) = \bar{v}_{i}^{0}$ for $i = 1,2,\dots,n$; $S_{A'}^{0}(\bar{l}_{\mathcal{L}}) = \bar{v}_{0}^{\mathcal{L}}$ and $S_{A'}^{0}(\bar{l}_{\mathcal{R}}) = \bar{v}_{0}^{\mathcal{R}}$. That is, the original agents are placed into the initial vertices while the leading agents are placed in a way that original agents can join either of them. The **goal arrangement** is: $S_{A'}^{+}(\bar{c}_i) = \bar{v}_{\vartheta,i}^{+}$ for $i = 1,2,\dots,n$; $S_{A'}^{+}(\bar{l}_{\mathcal{L}}) = \bar{v}_{-2}^{\mathcal{L}}$ and $S_{A'}^{+}(\bar{l}_{\mathcal{R}}) = \bar{v}_{-2}^{\mathcal{R}}$; that is, the original agents should finally reach the last row of the array part of the destination vertices and the leading agents should go out of the way.

The required conjugation of agents into the left and right vertices at a certain time step can be satisfied if the agents move a follows: all the agents $\bar{c}_1, \bar{c}_2, \dots, \bar{c}_n$

from the set of vertices $\bar{v}_1^0, \bar{v}_2^0, \ldots, \bar{v}_n^0$ move into the set of vertices $\bar{v}_1^{\mathcal{L}}, \bar{v}_2^{\mathcal{L}}, \ldots, \bar{v}_n^{\mathcal{L}}$ if the left branch is chosen, or into the set of vertices $\bar{v}_1^{\mathcal{R}}, \bar{v}_2^{\mathcal{R}}, \ldots, \bar{v}_n^{\mathcal{R}}$ if the right branch is chosen.

Without loss of generality, suppose the left branch has been chosen. Then agents $\bar{c}_1, \bar{c}_2, \ldots, \bar{c}_n$ together with the leading agent $\bar{l}_{\mathcal{L}}$ moves into vertices $\bar{v}_1^+, \bar{v}_2^+, \ldots, \bar{v}_n^+, \{\bar{v}_{\mathcal{R}}^+, \bar{v}_0^{\mathcal{R}}\}$. This is followed by movement of agents $\bar{c}_1, \bar{c}_2, \ldots, \bar{c}_n$ towards the last row of the array part of the destination vertices where their order is eventually **reversed** (if the right branch has been chosen no reversing is necessary). Leading agents return to their goal locations in $\bar{v}_{-2}^{\mathcal{L}}$ and $\bar{v}_{-2}^{\mathcal{R}}$ at the same time. The described behavior of agents within the optimal solution is ensured by locking appropriate vertices at appropriate time steps. That is, the pCPF instance Ξ is further extended with additional agents and vertices used for locking vertices as it is shown in the proof of Corollary 1. However, for sake of simplicity the description below will be restricted on the original components of the problem Ξ.

Thus, the optimal solution for the **left branch** $\mathcal{S}_{\mathcal{L},A'}^*(\Xi) = [S_{\mathcal{L},A'}^0, S_{\mathcal{L},A'}^1, \ldots, S_{\mathcal{L},A'}^{\zeta}]$ should satisfy that $S_{\mathcal{L},A'}^0(\bar{c}_i) = \bar{v}_i^0$, $S_{\mathcal{L},A'}^1(\bar{c}_i) = \bar{v}_i^{\mathcal{L}}$, $S_{\mathcal{L},A'}^2(\bar{c}_i) = \bar{v}_{i-1}^{\mathcal{L}}$, …, $S_{\mathcal{L},A'}^i(\bar{c}_i) = \bar{v}_1^{\mathcal{L}}$, $S_{\mathcal{L},A'}^{i+1}(\bar{c}_i) = \bar{v}_0^{\mathcal{L}}$, $S_{\mathcal{L},A'}^{i+2}(\bar{c}_i) = \bar{v}_{\mathcal{L}}^+$, $S_{\mathcal{L},A'}^{i+3}(\bar{c}_i) = \bar{v}_1^+$, $S_{\mathcal{L},A'}^{i+4}(\bar{c}_i) = \bar{v}_2^+$, …, $S_{\mathcal{L},A'}^{n+2}(\bar{c}_i) = \bar{v}_{n-i}^+$, $S_{\mathcal{L},A'}^{n+3}(\bar{c}_i) = \bar{v}_{n-i+1}^+$, $S_{\mathcal{L},A'}^{n+4}(\bar{c}_i) = \bar{v}_{1,n-i+1}^+$, $S_{\mathcal{L},A'}^{n+5}(\bar{c}_i) = \bar{v}_{2,n-i+1}^+$, …, $S_{\mathcal{L},A'}^{n+\vartheta+2}(\bar{c}_i) = \bar{v}_{\vartheta-1,n-i+1}^+$, and $S_{\mathcal{L},A'}^{n+\vartheta+3}(\bar{c}_i) = \bar{v}_{\vartheta,i}^+ = S_{A'}^+(\bar{c}_i)$ for $i = 1,2, \ldots, n$; $S_{\mathcal{L},A'}^0(\bar{l}_{\mathcal{L}}) = \bar{v}_0^{\mathcal{L}}$, $S_{\mathcal{L},A'}^1(\bar{l}_{\mathcal{L}}) \in \{\bar{v}_0^{\mathcal{L}}, \bar{v}_{\mathcal{L}}^+\}$, $S_{\mathcal{L},A'}^2(\bar{l}_{\mathcal{L}}) \in \{\bar{v}_{\mathcal{L}}^+, \bar{v}_1^+\}$, $S_{\mathcal{L},A'}^3(\bar{l}_{\mathcal{L}}) \in \{\bar{v}_1^+, \bar{v}_2^+\}$, …, $S_{\mathcal{L},A'}^{n+1}(\bar{l}_{\mathcal{L}}) \in \{\bar{v}_{n-1}^+, \bar{v}_n^+\}$, $S_{\mathcal{L},A'}^{n+2}(\bar{l}_{\mathcal{L}}) \in \{\bar{v}_n^+, \bar{v}_{\mathcal{R}}^+\}$, $S_{\mathcal{L},A'}^{n+3}(\bar{l}_{\mathcal{L}}) \in \{\bar{v}_{\mathcal{R}}^+, \bar{v}_0^{\mathcal{R}}\}$ (the left leading agent is going in front of the queue formed by the sequence of agents $\bar{c}_1, \bar{c}_2, \ldots, \bar{c}_n$), there is no special requirement on $S_{\mathcal{L},A'}^{n+4}(\bar{l}_{\mathcal{L}})$, $S_{\mathcal{L},A'}^{n+5}(\bar{l}_{\mathcal{L}})$, …, $S_{\mathcal{L},A'}^{n+\vartheta+2}(\bar{l}_{\mathcal{L}})$, indeed $S_{\mathcal{L},A'}^{n+\vartheta+3}(\bar{l}_{\mathcal{L}}) = \bar{v}_{-2}^{\mathcal{L}} = S_{A'}^+(\bar{l}_{\mathcal{L}})$. Similarly, there is no special requirement on $S_{\mathcal{L},A'}^i(\bar{l}_{\mathcal{R}})$ for any $i = 1,2, \ldots, n$. The optimal solution for the **right branch** $\mathcal{S}_{\mathcal{R},A'}^*(\Xi) = [S_{\mathcal{R},A'}^0, S_{\mathcal{R},A'}^1, \ldots, S_{\mathcal{R},A'}^{\zeta}]$ has almost the same form. The only difference is that the final reversal of the agents $\bar{c}_1, \bar{c}_2, \ldots, \bar{c}_n$ to fit into the last row of the array part of destination vertices is not performed. Observe, that the time step at which **conjugation** occurs is $t = 1$.

Now, the task is to show that the described behavior is feasible and no other behavior can occur within any optimal solution. In other words, any optimal solution of the problem has either the form of the solution for the **left branch** or the solution for the **right branch**.

The first row of the array part of destination vertices, that is, vertices $\bar{v}_{1,1}^+, \bar{v}_{1,2}^+, \ldots, \bar{v}_{1,n}^+$, is locked (closed for entering) for all the time steps except the time step $n + 4$. At this time step all the agents $\bar{c}_1, \bar{c}_2, \ldots, \bar{c}_n$ are entering the array part of the destination vertices. Then they continue towards their goal locations and hence vertices $\bar{v}_{1,1}^+, \bar{v}_{1,2}^+, \ldots, \bar{v}_{1,n}^+$ can be locked again for the remaining time steps. The vertices $\bar{v}_{-1}^{\mathcal{L}}$ and $\bar{v}_{-1}^{\mathcal{R}}$ are locked for all the time steps except the time

step $n + \vartheta + 2$. Similarly, the initial vertices are locked for all the time steps except the time step 0.

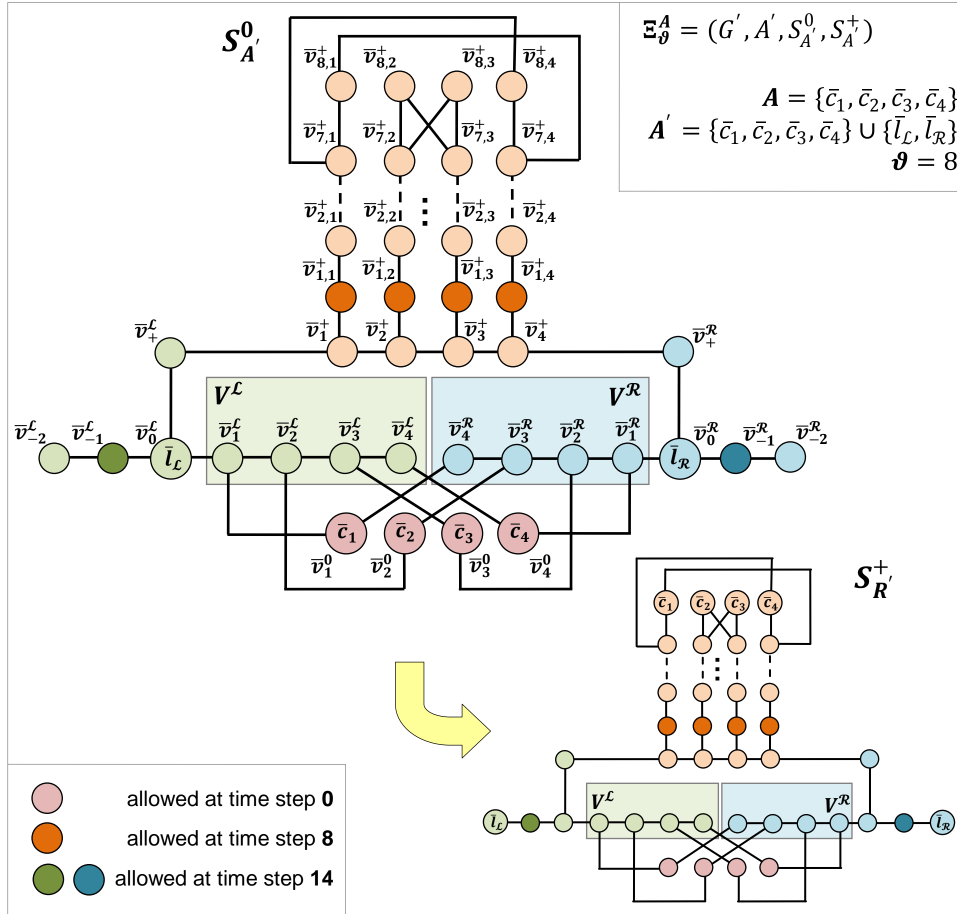


Figure 7. *A conjugation instance of the pCPF problem.* A conjugation instance Ξ_ϑ^A shown in the figure is constructed with respect to a set of agents $A = \{\bar{c}_1, \bar{c}_2, \bar{c}_3, \bar{c}_4\}$ and a parameter $\vartheta = 8$. The agents are restricted in their movements using vertex locking - namely, the initial vertices $\bar{v}_1^0$, $\bar{v}_2^0$, $\bar{v}_3^0$, and $\bar{v}_4^0$ can be entered only at time step 0; the vertices $\bar{v}_{1,1}^+$, $\bar{v}_{1,2}^+$, $\bar{v}_{1,3}^+$, and $\bar{v}_{1,4}^+$ can be entered only at time step 8; and the vertices $\bar{v}_{-1}^\mathcal{L}$ and $\bar{v}_{-1}^\mathcal{R}$ can be entered only at time step 14. These conditions enforce that the agents $\bar{c}_1$, $\bar{c}_2$, $\bar{c}_3$, and $\bar{c}_4$ are located either in vertices $\bar{v}_1^\mathcal{L}$, $\bar{v}_2^\mathcal{L}$, $\bar{v}_3^\mathcal{L}$, and $\bar{v}_4^\mathcal{L}$ or in vertices $\bar{v}_1^\mathcal{R}$, $\bar{v}_2^\mathcal{R}$, $\bar{v}_3^\mathcal{R}$, and $\bar{v}_4^\mathcal{R}$ at time step 1 in any optimal solution of Ξ_ϑ^A.

At the time of opening the first row of the array part of the destination vertices (at the time step $n + 3$), all the agents $\bar{c}_1, \bar{c}_2, \ldots, \bar{c}_n$ must reside in the vertices $\bar{v}_1^+, \bar{v}_2^+, \ldots, \bar{v}_n^+$ (eventually in the reversed order). Otherwise, they have no chance to reach their goal locations at all. Then, the fastest way to reach their goal locations starting from vertices $\bar{v}_1^+, \bar{v}_2^+, \ldots, \bar{v}_n^+$ is exact following shortest paths to the last row of the array part of destination vertices (all these paths are of the same

length). Since $\vartheta \geq n + 4$ which is enough time steps for the leading agents to reach their destination locations; the motion of agents $\bar{c}_1, \bar{c}_2, \dots, \bar{c}_n$ within the array part of the destination vertices represents the **bottleneck**.

It remains to check the behavior of agents before the time step $n + 3$. Since the initial vertices are allowed to be occupied only at time step 0, the agents $\bar{c}_1, \bar{c}_2, \dots, \bar{c}_n$ must enter the left or the right vertices immediately at the next time step. Between time steps 1 and $n + 3$ it is **not possible to swap** agents in the currently accessible part of the graph since it consists of a single path. Hence, if the agents $\bar{c}_1, \bar{c}_2, \dots, c_n$ split between the left and the right vertices, then they cannot be arranged into vertices $\bar{v}_1^+, \bar{v}_2^+, \dots, \bar{v}_n^+$ in the required order, because they are **obstructed** by the leading agents $\bar{l}_{\mathcal{L}}$ and $\bar{l}_{\mathcal{R}}$.

The just described instance will be called a ***conjugation*** *instance of the pCPF problem.* Notice, that the instance is parameterized by a set of agents A and an integer parameter $\vartheta \geq |A| + 4$. An instance of the problem corresponding to the given parameters will be denoted as Ξ_ϑ^A. Notice further, that the makespan of any optimal solution of Ξ_ϑ^A is $|A| + \vartheta + 3 \geq 2|A| + 7$. It is easy to see that the **size** of Ξ_ϑ^A is $(3 + \vartheta)|A| + 8$ which is $\mathcal{O}(\vartheta|A|)$.

An example of conjugation instance is shown in Figure 7. Although some edges of the conjugation instance intersect, it is just a matter of graph drawing in a plane. There is actually no interference between agents traversing edges that intersect (notice further that pCPF may take place in high dimensional spaces that cannot be drawn in plane without edge intersection).

3.4. *NP-completeness of pCPF*

All the ingredients are now prepared to prove that a *decision version* of the optimal pCPF is NP-complete. The **membership** into ***NP*** will be checked first. Then a **polynomial time reduction** of a *propositional satisfiability* instance (***SAT***) [1] to the instance of the decision version of the optimal pCPF will be constructed.

Definition 3 ***(decision version of pCPF).*** A *decision version of the optimal pCPF* is a task to decide for a given instance of pCPF Σ and a number $\eta \in \mathbb{N}_0$ whether there exists a solution $\mathcal{S}_A(\Sigma)$ of the makespan at most η. A notation Σ/η will be used for the decision instance. Next, let $pCPF_{OPT}$ denote the language of positive instances of this problem. □

It is not that easy to see that $pCPF_{OPT} \in NP$, since no upper bound on the size of the solution of $pCPF_{OPT}$ has been established so far. Hence, the standard technique of “guessing and checking” cannot be used immediately. Notice that, decision variants of several related *sliding piece problems* [10] such as ***Sokoban*** *game* [4] and ***Rush-hour*** *puzzle* [7] are proven to be $PSPACE$-complete [8, 9] but it is

not known whether they are in NP. The reason is that the polynomial upper bound on the size of the solution has not been found so far. Fortunately, this is not the case of $pCPF_{OPT}$. It is possible to establish the polynomial upper bound on the size of the solution of $pCPF_{OPT}$ using results shown in [13].

Lemma 3. $pCPF_{OPT} \in NP$. ■

Proof. It has been shown in [13] that there exists a solution $\mathcal{S}_P(\Pi) = [S_P^0, S_P^1, \ldots, S_P^{\xi}]$ for any solvable instance of the problem of PMG $\Pi = (G = (V,E), P, S_P^0, S_P^+)$ such that $\xi \in \mathcal{O}(|V|^3)$ (ξ is regarded as a function of Π here). Since the solution of an instance of PMG can be used as a solution of the corresponding pCPF instance (Proposition 1) it implies that there exists a solution $\mathcal{S}_A(\Sigma) = [S_A^0, S_A^1, \ldots, S_A^{\zeta}]$ for any solvable instance of the problem of pCPF $\Sigma = (G = (V,E), A, S_A^0, S_A^+)$ such that $\zeta \in \mathcal{O}(|V|^3)$ (ζ is regarded as a function of Σ as well). An instance of $pCPF_{OPT}$ Σ/η can be solved on a Turing machine with oracle in polynomial time as follows. A solution of the size $\mathcal{O}(|V|^3)$ of Σ is generated first by the oracle. Then, the generated solution is checked whether its makespan is at most η and whether it satisfies Definition 2. This check can be carried out in polynomial time with respect to the size of Σ/η. ■

Propositional satisfiability is a decision problem where the question is whether a given propositional formula has a satisfying valuation or not. As it is usual, propositional formulas in *conjunctive normal form* (*CNF*) [12] are considered. Let SAT denote the language of satisfiable instances of propositional formulas in CNF as it is formalized in the following definition.

Definition 4 ***(propositional satisfiability - SAT).*** A *propositional variable* is a variable that can be assigned either $TRUE$ or $FALSE$. A *literal* is propositional variable or its negation. A *clause* is disjunction of literals; that is, $\bigvee_{i=1}^{n} l_i$ where $n \in \mathbb{N}_0$ and l_i is a literal for $i = 1,2, \ldots, n$. A *propositional formula in CNF* is conjunction of clauses; that is, $\bigwedge_{j=1}^{m} C_j$ where $m \in \mathbb{N}_0$ and C_j is a clause for $i = 1,2, \ldots, m$. Let $Var(F)$ denote the set of propositional variables of CNF formula F, then *valuation* of variables of F is an assignment $e: Var(F) \longrightarrow \{FALSE, TRUE\}$. Valuation of variables is naturally extended from variables to literals, clauses, and the complete CNF formula. *Propositional satisfiability* problem (SAT) is a decision problem where the question is whether a given formula F in CNF has valuation of its variables so that F evaluates to $TRUE$ under this valuation. □

If CNF formula F has satisfying valuation, then F is said to be *satisfiable* otherwise it is said to be *unsatisfiable* (examples of several propositional formulae in

CNF are shown in Figure 8). It is well known that SAT is NP-complete. However, a slight technical adaptation of propositional satisfiability is necessary to carry out the required reduction to pCPF. A restriction on formulas in CNF where positive and negative *literals* of the same variable have the same number of occurrences in the formula will be made. Let the language of satisfiable formulas that comply with this restriction will be denoted as $SAT_=$ (see Figure 8 again).

Definition 5 ***(equality propositional satisfiability - $SAT_=$).*** Let F be a propositional formula in CNF. Next, let $pos(x,F)$ with $x \in Var(F)$ denote the set of positive occurrences of x in F and similarly let $neg(x,F)$ denote a set of negative occurrences of x in F. *Equality propositional satisfiability problem* ($SAT_=$) is a decision problem where the question is whether a given formula F in CNF such that $|pos(x,F)| = |neg(x,F)|$ for every $x \in Var(F)$ is satisfiable or not. □

Lemma 4. $SAT_=$ is NP-complete. ■

Proof. With respect to the membership into NP, the restriction makes no change; thus $SAT_= \in NP$. Any instance of SAT can be reduced to an instance of $SAT_=$ by adding *clauses* to balance the number of positive and negative literals of the same variable. The added clauses should preserve equisatisfiability of the resulting formula with the original one. Let F is a formula in CNF and let x be a variable with **unbalanced** positive and negative occurrences. Without loss of generality let $|pos(x,F)| < |neg(x,F)|$. Then a clause $(\bigvee_{i=1}^{|neg(x,F)|-|pos(x,F)|} x) \vee y \vee \neg y$ where y is a new variable is added to F. Now x as well as newly added y have the same number of positive and negative occurrences. Clearly, the resulting formula is equisatisfiable with F since the newly added clause is always satisfied. The described process should be done for all the unbalanced variables. The length of the resulting formula is at most twice of F, thus the reduction can be done in polynomial time. ■

Theorem 1. $pCPF_{OPT}$ is NP-complete. ■

Proof. It remains to prove that $pCPF_{OPT}$ is NP-hard. A polynomial time reduction of $SAT_=$ to $pCPF_{OPT}$ will be used. Let $F_=$ be a formula in CNF, that is, $F_= = \bigwedge_{i=1}^{n}(\bigvee_{j}^{k_i} l_j^i)$, where l_j^i is jth literal of ith clause; there are n clauses, where ith clause has k_i literals.

Assume further that that each variable has the same number of positive and negative occurrences in $F_=$. Let $Var(F_=)$ denote the set of propositional variables of $F_=$. An instance $\Sigma = (G = (V,E), A, S_A^0, S_A^+)/\eta$ of the decision version of the optimal pCPF for $F_=$ will be constructed in the following way. Every occurrence

of a literal in $F_=$ will be associated with a vertex. Thus, a set of vertices $V^{F_=} = \bigcup_{i=1}^{n} \bigcup_{j=1}^{k_i} \{\bar{l}_j^i\}$ is constructed ($\bar{l}_j^i$ is a symbol while l_j^i is a variable standing for a literal); a vertex $\bar{l}_j^i$ corresponds to an occurrence of a literal l_j^i in ith clause as jth disjunct. A **conjugation** instance of pCPF will be associated with each propositional variable of $F_=$ while left and right vertices of the conjugation graph will be one-to-one matched to vertices from $V^{F_=}$ that correspond to negative and positive occurrences of the variable respectively. This is possible since there is the same number of positive and negative occurrences of each variable in $F_=$ (conjugation graph has the same number of left and right vertices).

$$\boldsymbol{F_1} = \underbrace{(\neg x_1 \vee \neg x_2)}_{C_1} \wedge \underbrace{x_1}_{C_2} \wedge \underbrace{x_2}_{C_3}$$

unsatisfiable

$Var(F_1) = \{x_1, x_2\}$ $\quad pos(x_1, F_1) = \{C_2\}$ $\quad neg(x_1, F_1) = \{C_1\}$

$pos(x_2, F_1) = \{C_3\}$ $\quad neg(x_2, F_1) = \{C_1\}$

$$\boldsymbol{F_2} = \underbrace{(x_1 \vee \neg x_2 \vee x_3)}_{C_1} \wedge \underbrace{(\neg x_1 \vee \neg x_2 \vee \neg x_3)}_{C_2}$$

satisfiable with $e(x_1) = TRUE, e(x_2) = FALSE, e(x_3) = TRUE$

$Var(F_2) = \{x_1, x_2, x_2\}$ $\quad pos(x_1, F_2) = \{C_1\}$ $\quad neg(x_1, F_2) = \{C_2\}$

$pos(x_2, F_2) = \emptyset$ $\quad neg(x_2, F_2) = \{C_1, C_2\}$

$pos(x_3, F_2) = \{C_1\}$ $\quad neg(x_3, F_2) = \{C_2\}$

$$\boldsymbol{F_3} = \underbrace{(x_1 \vee \neg x_2 \vee x_3)}_{C_1} \wedge \underbrace{(\neg x_1 \vee \neg x_2 \vee \neg x_3)}_{C_2} \wedge \underbrace{(x_2 \vee x_2 \vee y_1 \vee \neg y_1)}_{C_3}$$

satisfiable with $e(x_1) = TRUE, e(x_2) = FALSE, e(x_3) = TRUE, e(y_1) = TRUE$

$Var(F_3) = \{x_1, x_2, x_2, y_1\}$ $\quad pos(x_1, F_3) = \{C_1\}$ $\quad neg(x_1, F_3) = \{C_2\}$

$pos(x_2, F_3) = \{C_3, C_3{}'\}$ $\quad neg(x_2, F_3) = \{C_1, C_2\}$

$pos(x_3, F_3) = \{C_1\}$ $\quad neg(x_3, F_3) = \{C_2\}$

$pos(y_1, F_3) = \{C_3\}$ $\quad neg(y_1, F_3) = \{C_3\}$

Figure 8. ***Examples*** *of propositional formulae in* ***CNF***. Three formulae F_1, F_2, and F_3 are shown. F_1 is unsatisfiable while other two are satisfiable. Positive and negative occurrences of literals in formulae are depicted. Notice that the number of positive and negative occurrences of x_2 in F_2 is unbalanced; that is, $F_2 \notin SAT_=$ (satisfiable but syntactically incorrect). The result of rebalancing of F_2 is $F_3 \in SAT_=$ (both satisfiable and syntactically correct).

The idea is to prepare a group of agents of the size $|pos(x, F_=)| = |neg(x, F_=)|$ for each Propositional variable $x \in Var(F_=)$. This group of agents

will be placed in the initial vertices of the conjugation subgraph corresponding to x. The construction of the conjugation subgraph will enforce that all the agents must go either into the vertices corresponding to positive literals or into the vertices corresponding to negative literals. If the movement of agents is interpreted in the way that literals corresponding to vertices of $V^{F_=}$ visited at time step 1 will be assigned the same propositional value, then the conjugation technique assures **propositional consistency** of the assignment. However, this is not enough to establish correspondence between an assignment satisfying $F_=$ and a solution of Σ/η. It is furthermore necessary to make agents simulate **clause satisfaction** by any solution whose makespan is at most η.

This can be done by enforcing agents either to visit at least one literal/vertex of each clause of $F_=$ (in the case when visited literals/vertices are assigned the value $TRUE$) or leave at least one literal/vertex of each clause of $F_=$ unoccupied at time step 1 (in the case when visited literals/vertices are assigned the value $FALSE$). Since the second option can be easily implemented through the vertex **set locking** mechanism (Proposition 3, Corollary 4), the value $FALSE$ will be used for literals corresponding to vertices visited at time step 1.

Nevertheless, some technical details such as the exact specification of η need to be discussed. The equality between makespans of optimal solutions over the individual conjugation instances needs to be established.

Recall that a conjugation instance Ξ_{ϑ}^{A} is characterized by two parameters: A – the set conjugated agents and ϑ – parameter affecting the makespan of the optimal solution of the instance. Let $\varrho = max_{x \in Var(F_=)}\{|pos(x, F_=)| = |neg(x, F_=)|\}$. For a given $x \in Var(F_=)$, the conjugation instance $\Xi_{\vartheta(x)}^{A(x)}$ will have the parameters $A(x) = \{\bar{c}_1^x, \bar{c}_2^x, \ldots, \bar{c}_{|pos(x,F_=)|}^x\}$ and $\vartheta(x) = 2\varrho - |pos(x, F_=)| + 4 \geq |A(x)| + 4$. Hence, the makespan of any optimal solution of the conjugation instance $\Xi_{\vartheta(x)}^{A(x)}$ is $|A(x)| + \vartheta(x) + 3 = 2\varrho + 7$.

Matching of left and right vertices of $\Xi_{\vartheta(x)}^{A(x)}$ to vertices from $V^{F_=}$ is as follows: $\{\bar{v}_1^{x,\mathcal{L}}, \bar{v}_2^{x,\mathcal{L}}, \ldots, \bar{v}_{|A(x)|}^{x,\mathcal{L}}\} = \{\bar{l}_j^i \mid l_j^i = \neg x; i = 1,2,..n; j = 1,2,\ldots,k_i\}$ and $\{\bar{v}_1^{x,\mathcal{R}}, \bar{v}_2^{x,\mathcal{R}}, \ldots, \bar{v}_{|A(x)|}^{x,\mathcal{R}}\} = \{\bar{l}_j^i \mid l_j^i = x; i = 1,2,..n; j = 1,2,\ldots,k_i\}$. Now, the crucial observation has to be made. It holds that $F_=$ is satisfiable if and only if there **exists a solution** of the currently constructed instance of makespan of $2\varrho + 7$ such that at time step 1 at least one vertex from the set of vertices corresponding to each clause remains unoccupied.

Let $e: Var(x) \longrightarrow \{FALSE, TRUE\}$ be a satisfying valuation of $F_=$. If $e(x) = FALSE$, then agents $\bar{c}_1^x, \bar{c}_2^x, \ldots, \bar{c}_{|pos(x,F_=)|}^x$ are placed in $\bar{v}_1^{x,\mathcal{R}}, \bar{v}_2^{x,\mathcal{R}}, \ldots, \bar{v}_{|A(x)|}^{x,\mathcal{R}}$ at time step 1; if $e(x) = TRUE$ then they are placed in $\bar{v}_1^{x,\mathcal{L}}, \bar{v}_2^{x,\mathcal{L}}, \ldots, \bar{v}_{|A(x)|}^{x,\mathcal{L}}$ at time step 1.

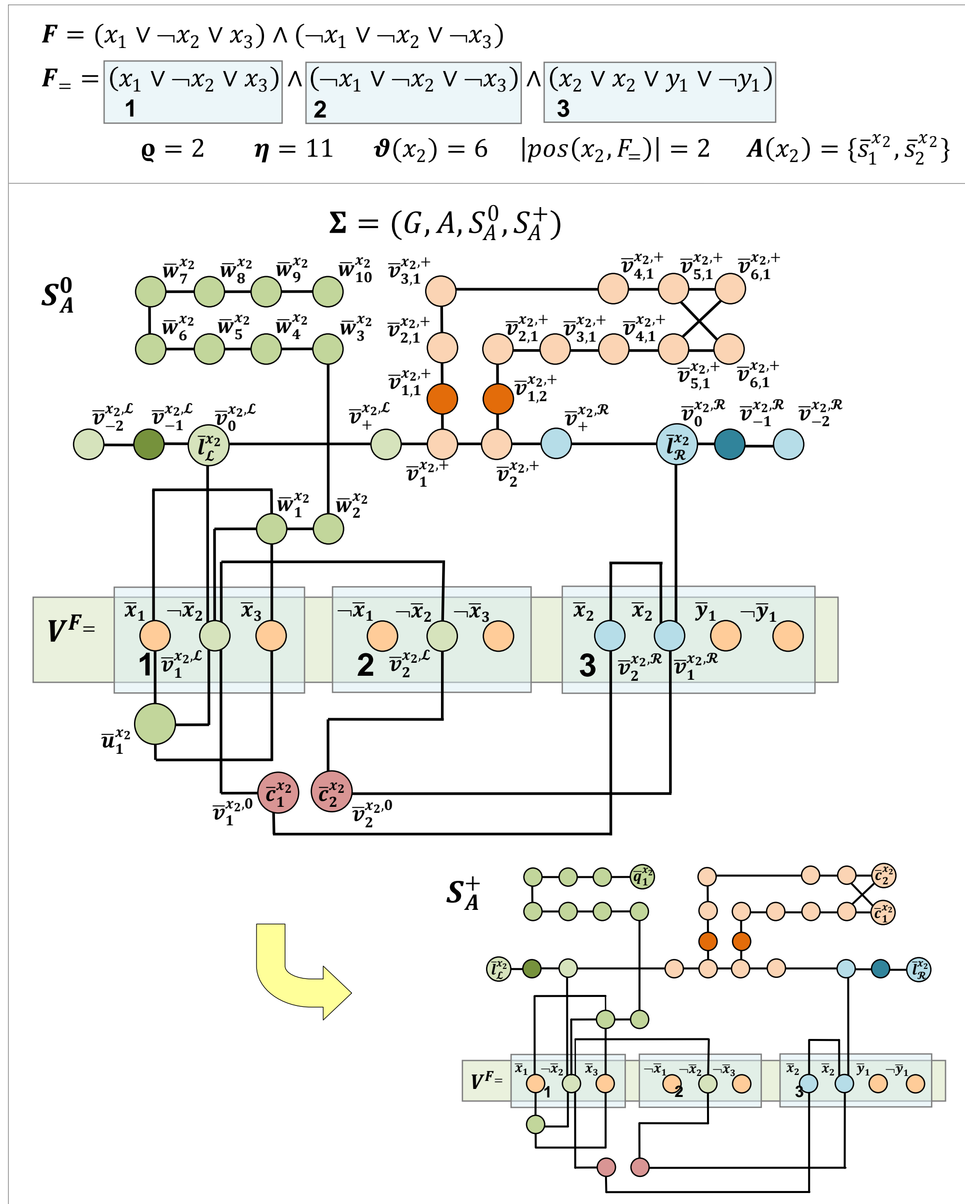


Figure 9. *A polynomial time* ***reduction*** *of a propositional formula to a decision instance of pCPF.* A formula F is transformed to a formula $F_=$ in which each variable has the same number of positive and negative occurrences. Then an instance of the decision version of the problem of pCPF Σ/η is constructed. The conjugation technique is used to simulate propositional consistency and the set locking technique is used to simulate clause satisfaction (the reduction of one variable using the conjugation technique and the reduction of one clause using the set locking technique are shown). There exists a solution of Σ of the makespan $\eta = 11$ if and only if the formula F is satisfiable.

The placement of agents at time steps other than 1 is straightforward. Since e is the satisfying assignment, at least one vertex from the set of vertices corresponding to each clause remains unoccupied. On the other hand, Corollary 4 can be used to augment the instance to enforce the at least one vertex from the set of vertices that corresponds to literals of a clause is not occupied by agents from the set $\bigcup_{x \in Var(F_=)} A(x)$ within **any optimal solution** while the makespan of $2\varrho + 7$ remains preserved. That is, Corollary 4 is invoked with $W = \{\overline{l}_1^i, \overline{l}_2^i, \dots, \overline{l}_{k_i}^i\}$ that corresponds to satisfying the ith clause of $F_=$. Let Σ denote the resulting instance. Any solution of the makespan of $2\varrho + 7$ of Σ satisfies conditions at time step 1 and hence it induces a satisfying assignment of $F_=$.

The construction of Σ requires **polynomial** time in size of $F_=$; the size of Σ is also polynomial (the size of each conjugation subgraph is polynomial in size of $F_=$ and the number of conjugation subgraphs is bounded by the size of $F_=$). Now, if Σ has a solution of the makespan $\eta = 2\varrho + 7$ then it is ensured that **conjugation** and **clause satisfaction** has been successfully simulated, thus a satisfying valuation of $F_=$ can be easily derived from this solution. Hence, $F_= \in SAT_=$ if and only if $\Sigma/\eta \in pCPF_{OPT}$. Together with Lemma 3 the claim that $pCPF_{OPT}$ is NP-complete has been obtained. ■

The reduction described in the proof is illustrated in Figure 9. The illustration shows instantiation of conjugation mechanism over single variable. Connection of simulation of clause satisfaction to conjugation mechanism is also shown in Figure 9.

4. Related Works and Conclusion

A **parallel** version of the cooperative path-finding problem (pCPF) is introduced in this paper. The new theoretical result shown in this paper is that the decision version of the optimal pCPF is ***NP*-complete**. The parameter, which is optimized, is the makespan, that is maximum of arrival times to destination over all the agents.

The **reduction** of propositional satisfiability to pCPF has been used for the proof of *NP*-hardness. Numerous techniques to simulate propositional consistency and clause satisfaction within pCPF were developed in this work. These techniques were inspired by works on multi-commodity flows [6]. We assume developed techniques generic enough to be used in different contexts. Recently vertex locking and conjugation techniques were used in the proof of *NP*-hardness of checking the existence of winning strategy in a so called *adversarial CPF* (ACPF) [119]. ACPF is CPF with multiple teams of agents, which compete in reaching their goals.

The fact that optimal pCPF is *NP*-complete is a quite negative result. Fortunately, if the requirement on the shortest possible makespan of solutions is relaxed, the problem becomes **tractable**. Namely, it belongs to the *P* class. However, the situation is not that straightforward. Although algorithms developed for solving PMG/CPF [13, 37] can be used for solving pCPF, this practice is disadvantageous. Despite a promising theoretical makespan of $\mathcal{O}(|V|^3)$ of solutions generated by these algorithms, the makespan measured empirically is relatively high [28] due to a large constant in the estimation. That is why alternative solving sub-optimal algorithms for pCPF producing better solutions (so called *BIBOX* algorithms) and solution improving techniques were proposed [23, 24, 25, 28]. Recently there has been considerable development in sub-optimal algorithms for CPF represented by works [121].

An important **related work** is represented by articles [30, 31, 32, 33]. The authors study another version of pCPF, which is similar to the version presented in this paper. The authors define the tractable class of this problem where graphs are restricted on grids and there is a relative abundance of unoccupied vertices.

Several attempts to solve the standard non-parallel CPF optimally has been made. An algorithm based on A* has been presented in [21]. The algorithm is suitable for CPF instances with few agents and lot of free space in the graphs. Different approach to solving CPF optimally is to translate CPF to SAT, which has been suggested in [121]. Interestingly SAT based methods seem to be complementary to A* based as they perform well on densely occupied instances.

An interesting question for future work is whether it is feasible to find a solution of a pCPF instance which is constantly worse than the optimum. Currently, it is an open question whether such an **approximation** algorithm exists. The answer to this question will consequently provide the estimation of how far from the optimum are the solutions generated by algorithms for the non-optimization case of the problem. Consequently, the estimation of what is the makespan of the optimal solution of large instances would be also available.

Glossary

PMG	pebble motion on a graph
$G = (V, E)$	an undirected graph; V denotes a set of vertices; E denotes set of edges
P	a set of pebbles
μ	the number of pebbles
$\bar{p}_i$	a pebble
$S_P^0: P \longrightarrow V$	the initial arrangement of pebbles
$S_P^+: P \longrightarrow V$	the goal arrangement of pebbles

$S_P^t: P \longrightarrow V$	the arrangement of pebbles at time step t
ξ	the makespan of a solution of PMG and a sequence of pebble arrangements
$\mathcal{S}_P$	a sequence of arrangements of pebbles forming a solution of PMG
$\Pi = (G, P, S_P^0, S_P^+)$	an instance of PMG
$\mathcal{S}_P(\Pi)$	a solution to the instance of PMG
CPF	cooperative path-finding
pCPF	parallel cooperative path-finding
A	a set of agents
ν	the number of agents
$\bar{a}_i$	an agent
$S_A^0: A \longrightarrow V$	the initial arrangement of agents
$S_A^+: A \longrightarrow V$	the goal arrangement of agents
$S_A^t: A \longrightarrow V$	the arrangement of agents at time step t
ζ	the makespan of a solution of pCPF
ζ^*	the makespan of an optimal solution of pCPF
$\mathcal{S}_A$	a sequence of arrangements of agents forming a solution of pCPF
$\Sigma = (G, A, S_A^0, S_A^+)$	an instance of pCPF
$\mathcal{S}_A(\Sigma)$	a solution to the instance of pCPF
$Sol(\Sigma)$	a set of solutions of a pCPF instance Σ
$Sol^*(\Sigma)$	a set of makespan optimal solutions of a pCPF instance Σ
Ξ_ϑ^A	a conjugation instance of pCPF; ϑ is the length of a solution
$pCPF_{OPT}$	a language consisting of pairs Σ/η where Σ is a pCPF solvable by solution of the makespan at most η
SAT	a language consisting of satisfiable propositional formulas in CNF
$SAT_=$	a subset of SAT where each variable has the same number of positive and negative occurrences
F	a propositional formula in CNF
$F_=$	a propositional formula in CNF where each variable has the same number of its positive and negative occurrences
$pos(x, F)$	a set of positive occurrences of propositional variable x in F
$neg(x, F)$	a set of negative occurrences of propositional variable x in F
$O\|_V$	the restriction of an object O on a set of vertices V
V_X	newly added vertices
E_X	newly added edges
A_X	newly added agents

References

1. R. K. **Ahuja**, T. L. **Magnanti**, and J. B. **Orlin**. *Network flows: theory, algorithms, and applications.* Prentice Hall, 1993, ISBN 978-0136175490.
2. S. A. **Cook**. *The Complexity of Theorem Proving Procedures.* Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971), pp. 151-158, ACM Press, 1971.
3. T. H. **Cormen**, C. E. **Leiserson**, R. L. **Rivest**, and C. **Stein**. *Introduction to Algorithms (Second edition).* MIT Press and McGraw-Hill, 2001, ISBN 0-262-03293-7.
4. J. C. **Culberson**. *Sokoban is PSPACE-complete.* Technical Report TR 97-02, Department of Computing Science, University of Alberta, 1997, http://webdocs.cs.ualberta.ca/~joe/Preprints/Sokoban/ index.html [accessed April 2010].
5. J. D. **Dixon** and B. **Mortimer**. *Permutation Groups*. in Graduate Texts in Mathematics, Volume 163, Springer, 1996, ISBN 978-0-387-94599-6.
6. S. **Even**, A. **Itai**, A. **Shamir**. *On the Complexity of Timetable and Multicommodity Flow Problems.* SIAM Journal on Computing, Volume 5 (4), pp. 691-703, Society for Industrial and Applied Mathematics, 1976.
7. G. W. **Flake**, E. B. **Baum**. *Rush Hour is PSPACE-complete, or "Why you should generously tip parking lot attendants".* Theoretical Computer Science, Volume 270(1-2), pp. 895-911 Elsevier, 2002.
8. M. R. **Garey** and D. S. **Johnson**. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1979, ISBN: 978-0716710455.
9. R. A. **Hearn** and E. D. **Demaine**. *PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation.* Theoretical Computer Science, Volume 343(1-2), pp. 72-96, Elsevier, 2005.
10. E. **Hordern**. *Sliding Piece Puzzles.* Oxford University Press, 1986, ISBN: 978-0198532040.
11. M. **Ivanová**, P. **Surynek**. *Adversarial Cooperative Path-Finding: A First View.* The 27th AAAI Conference on Artificial Intelligence (AAAI 2013), Bellevue, WA, USA, late breaking track, technical report, AAAI Press, 2013
12. P. **Jackson**, D. **Sheridan**. *Clause Form Conversions for Propositional Circuits.* Theory and Applications of Satisfiability Testing, 7th International Conference (SAT 2004), Revised Selected Papers, pp. 183–198, Lecture Notes in Computer Science 3542, Springer 2005.
13. D. **Kornhauser**, G. L. **Miller**, and P. G. **Spirakis**. *Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications*. Proceedings of the 25th Annual Symposium on Foundations of Computer Science (FOCS 1984), pp. 241-250, IEEE Press, 1984.

14. C. H. **Papadimitriou**, P. **Raghavan**, M. **Sudan**, and H. **Tamaki**. Motion Planning on a Graph. Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS 1994), pp. 511-520, IEEE Press, 1994.

15. D. **Ratner** and M. K. **Warmuth**. *Finding a Shortest Solution for the N×N Extension of the 15-PUZZLE Is Intractable*. Proceedings of the 5th National Conference on Artificial Intelligence (AAAI 1986), pp. 168-172, Morgan Kaufmann Publishers, 1986.

16. D. **Ratner** and M. K. **Warmuth**. *NxN Puzzle and Related Relocation Problems.* Journal of Symbolic Computation, Volume 10 (2), pp. 111-138, Elsevier, 1990.

17. M. R. K. **Ryan**. *Graph Decomlocation for Efficient Multi-Robot Path Planning*. Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 2003-2008, IJCAI Conference, 2007.

18. M. R. K. **Ryan**. *Exploiting Subgraph Structure in Multi-Robot Path Planning*. Journal of Artificial Intelligence Research (JAIR), Volume 31, pp. 497-542, AAAI Press, 2008.

19. P. E. **Schupp** and R. C. **Lyndon**. *Combinatorial group theory*. Springer, 2001, ISBN 978-3-540-41158-1.

20. D. **Silver**. *Cooperative Pathfinding.* Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2005), pp. 117-122, AAAI Press.

21. T. **Standley**. *Finding Optimal Solutions to Cooperative Pathfinding Problems.* Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010), pp. 173-178, AAAI Press, 2010.

22. P. **Surynek**. *A Novel Approach to Path-finding for Multiple Agents in Bi-connected Graphs*. Proceedings of the 2009 IEEE International Conference on Agentics and Automation (ICRA 2009), pp. 3613-3619, IEEE Press, 2009.

23. P. **Surynek**. *Towards Shorter Solutions for Problems of Path-finding for Multiple Agents in θ-like Environments*. Proceedings of the 22nd International FLAIRS Conference (FLAIRS 2009), pp. 207-212, AAAI Press, 2009.

24. P. **Surynek**. *Making Solutions of Cooperative path-finding Problems Shorter Using Weak Translocations and Critical Path Parallelism*. Proceedings of the 2009 International Symposium on Combinatorial Search (SoCS 2009), University of Southern California, 2009, http://www.search-conference.org/index.php/Main/SOCS09 [accessed July 2009].

25. P. **Surynek**. *An Application of Pebble Motion on Graphs to Abstract Cooperative path-finding*. Proceedings of the 21st International Conference on Tools with Artificial Intelligence (ICTAI 2009), pp. 151-158, IEEE Press, 2009.

26. P. **Surynek**. *An Optimization Variant of Cooperative path-finding is Intractable.* Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010), pp. 1261-1263, AAAI Press, 2010.

27. P. **Surynek**. *Towards Optimal Cooperative Path Planning in Hard Setups through Satisfiability Solving*. Proceedings of 12th Pacific Rim International Conference on Artificial Intelligence (PRICAI 2012), pp. 564-576, LNCS 7458, Springer, 2012.
28. P. **Surynek**. *Solving Abstract Cooperative Path-Finding in Densely Populated Environments.* Computational Intelligence, Volume 30, Issue 2, pp. 402-450, Wiley, 2014.
29. R. E. **Tarjan**. *Depth-First Search and Linear Graph Algorithms.* SIAM Journal on Computing, Volume 1 (2), pp. 146-160, Society for Industrial and Applied Mathematics, 1972.
30. K. C. **Wang** and A. **Botea**. *Tractable Cooperative path-finding on Grid Maps*. Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009), pp. 1870-1875, IJCAI Conference, 2009.
31. K. C. **Wang**. *Bridging the Gap between Centralised and Decentralised Multi-Agent Pathfinding.* Proceedings of the 14th Annual AAAI/SIGART Doctoral Consortium (AAAI-DC 2009), pp. 23-24, AAAI Press, 2009.
32. K. C. **Wang** and A. **Botea**. *Fast and Memory-Efficient Multi-Agent Pathfinding.* Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008), Australia, pp. 380-387, AAAI Press, 2008, ISBN 978-1-57735-386-7.
33. K. C. **Wang** and A. **Botea**. *Scalable Multi-Agent Pathfinding on Grid Maps with Tractability and Completeness Guarantees*. Proceedings of the European Conference on Artificial Intelligence (ECAI 2010), IOS Press, 2010.
34. D. B. **West**. *Introduction to Graph Theory.* Prentice Hall, 2000, ISBN: 978-0130144003.
35. J. **Westbrook**, R. E. **Tarjan**. *Maintaining bridge-connected and biconnected components on-line*. Algorithmica, Volume 7, Number 5&6, pp. 433–464, Springer, 1992.
36. B. de **Wilde**, A. ter **Mors**, C. **Witteveen**. *Push and rotate: cooperative multi-agent path planning*. Proceedings of International conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2013), pp. 87-94, IFAAMAS, 2013.
37. R. M. **Wilson**. *Graph Puzzles, Homotopy, and the Alternating Group.* Journal of Combinatorial Theory, Ser. B 16, pp. 86-96, Elsevier, 1974.

Pavel **Surynek**, Petra **Surynková**, Miloš **Chromý**: *The Impact of a Bi-connected Graph Decomposition on Solving Cooperative Path-finding Problems.* Fundamenta Informaticae, Volume 135 (3), pp. 295-308, IOS Press, 2014.

The Impact of a Bi-connected Graph Decomposition on Solving Cooperative Path-finding Problems

Pavel Surynek[*], **Petra Surynková**[**], and **Miloš Chromý**[*]

Charles University in Prague
Faculty of Mathematics and Physics

[*] *Malostranské náměstí 25, 118 00 Praha 1, Czech Republic*
[**] *Sokolovská 83, 186 75 Praha 8, Czech Republic*

pavel.surynek@mff.cuni.cz, petra.surynkova@mff.cuni.cz, miloschromy@gmail.com

Abstract. This paper proposes a framework for analyzing algorithms for inductive processing of bi-connected graphs. The *BIBOX* algorithm for solving cooperative path-finding problems over bi-connected graphs is submitted for the suggested analysis. The algorithm proceeds according to a decomposition of a given bi-connected graph into handles. After finishing a handle, the handle is ruled out of consideration and the processing task is reduced to a task of the same type on a smaller graph. The handle decomposition for which the *BIBOX* algorithm performs best is theoretically identified. The conducted experimental evaluation confirms that the suggested theoretical analysis well corresponds to the real situation.

Keywords**:** BIBOX algorithm, bi-connected graphs, cooperative path-finding, complexity

1. Introduction and Motivation

Many graph-processing algorithms proceed inductively. That is, after processing a part of the graph, the processing task is reduced to a task of the same type on a smaller graph where the finished part is no longer considered. Classical graph algorithms for *finding minimum spanning trees* (*Borůvka's* and *Jarník's* algorithms [1, 3, 5, 6, 7]) and *single source shortest paths* (*Dijkstra's* algorithm [2]) proceed exactly in this manner. In the former case, the induction step typically

consumes an edge, while in the latter case, the induction step implies the processing of a vertex.

We would like to focus on another problem that takes place on a graph, namely the *cooperative path-finding problem* (CPF) [1, 8, 9, 14] where the task is to relocate certain objects (*agents*) along the edges of a given graph in a non-colliding way in order to reach the given goal vertices. An algorithm called *BIBOX* for solving CPF on *bi-connected* graphs was introduced in [10, 11]. It is another representative of an induction-based algorithm. The given bi-connected graph is processed inductively according to its decomposition into handles (a path connected by its two ends to the rest of the graph) by the algorithm. Here, the induction step is represented by handle processing. However, the analysis given in [11] does not address how the handle decomposition affects the overall performance of the algorithm. The objective of this paper is to fill this gap and determine the impact of the choice of a particular handle decomposition on the performance of the *BIBOX* algorithm.

A general framework for analyzing algorithms processing bi-connected graphs inductively is proposed. The *BIBOX* algorithm is subjected to the analysis with the view of finding handle decompositions on which the algorithm performs best. An experimental evaluation is conducted to verify the theoretical findings.

2. Background

Bi-connected graphs arise in many real-life scenarios dealing with navigation networks as they capture the important property that at least two alternative (disjoint) routes connect any two vertices. In other words, any two vertices lie on a cycle. This property can be well utilized in CPF, as it allows for relocations around cycles (rotations).

Definition 1 ***(bi-connected graph).*** An undirected graph $G = (V, E)$ is *bi-connected* if $|V| \geq 3$ and the graph after deletion of any vertex is connected; that is, the graph $G' = (V', E')$, where $V' = V \setminus \{v\}$ and $E' = \{\{u, w\} | u, w \in V \wedge u \neq v \wedge w \neq v\}$, is connected for every $v \in V$. □

Observe that a cycle itself is a bi-connected graph according to the aforementioned definition. The well known property that is utilized by algorithms for processing bi-connected graphs is that any bi-connected graph can be inductively build by adding handles [12, 13]. The operation of adding the handle $L = [u, w_1, w_2, \ldots, w_l, v]$ with $l \in \mathbb{N}_0$, which is a sequence of vertices, to the graph $G = (V, E)$ so that $\{u, w_1, w_2, \ldots, w_l, v\} \cap V = \{u, v\}$ (that is, the handle consists

of already present vertices u, v and fresh vertices w_j for $j = 1,2, \dots, l$) results in a new graph $G' = (V', E')$, where $V' = V \cup \{w_1, w_2, \dots, w_l\}$ and $E' = E \cup \{\{u, w_1\}, \{w_2, w_3\}, \dots, \{w_{l-1}, w_l\}, \{w_l, v\}\}$. The *size of the handle* L is defined as l, that is the number of internal vertices. The addition of a handle of size 0 corresponds to the addition of an edge.

Proposition 1 ***(handle decomposition)*** [12, 13]**.** Any bi-connected $G = (V, E)$ graph can be obtained from a cycle using a sequence of handle-adding operations. ■

The sequence of handles together with the initial cycle, from which the given bi-connected graph can be constructed, will be called a *handle decomposition*. The handle decomposition for a given bi-connected graph $G = (V, E)$ will be denoted as $C_0, L_1, L_2, \dots, L_k$, where C_0 is a cycle (formally a sequence of vertices) and L_i is a handle for $i = 1,2, \dots, k$ (for formal definition of sequences of vertices see above). G can be reconstructed from $C_0, L_1, L_2, \dots, L_k$ as follows: first cycle C_0 is laid; then the handles $L_1, L_2, \dots, L_k$ are added inductively. Consider that we have an intermediate graph G_i (as a result of adding the handles $L_1, L_2, \dots, L_i$), the next intermediate graph G_{i+1} is obtained by adding L_{i+1} to G_i using the handle-adding operation. The final intermediate graph G_k is G. Notice that $V = C_0 \cup L_1 \cup L_2 \cup \dots \cup L_k$ and the intermediate graph at any step of the process of reconstruction from handles is bi-connected.

3. Theoretical Analysis of Induction-based Algorithms and a Special Case

Assume that the algorithm processes handle decomposition handles one by one starting with the last one and moving towards the initial cycle. After the processing of a handle is finished, the algorithm continues inductively on a smaller bi-connected graph without the finished handle. Let $t(n, m)$ be the upper bound of the time consumed by the algorithm when processing a handle of size n in a bi-connected graph of size $n + m$.

Particularly in the case of the *BIBOX* algorithm that solves the cooperative path-finding problem it holds that $(n, m) = d_1 \cdot n^3 + d_2 \cdot n^2 m + d_3 \cdot n m^2$, where $d_1, d_2, d_3 \in \mathbb{R}$.

The upper bound of the time needed to process the given bi-connected graph $G = (V, E)$ according to the handle decomposition $V = C_0 \cup L_1 \cup L_2 \cup \dots \cup L_k$ can thus be calculated as follows:

$$\sum_{i=1}^{k} t\left(|L_i|, |C_0| + \sum_{j=1}^{i-1} |L_j|\right) \tag{1}$$

Obviously, the expression depends on the number of handles in the decomposition as well as on the sizes of the individual handles. Several scenarios with different distributions of handle sizes will be evaluated.

3.1. *Varying the Number and the Size of Handles*

The number of handles may vary in multiple different ways. In practice, however, cases with sizes of the handle L_i that can be expressed as $\mathcal{O}(1)$, $\mathcal{O}(i)$, $\mathcal{O}(i^2)$, $\mathcal{O}(\sqrt{i})$, or $\mathcal{O}(2^i)$ can be expected as a result of constructing a handle decomposition satisfying certain constraints. It is also natural to expect that the size of handles is a growing function with respect to their position in the handle decomposition (the larger part of the graph allows a larger handle to be present) with which the suggested handle sizes comply.

Assume that the size of the handle depends linearly on its position within the handle decomposition, that is, $|L_i| = \alpha i$; $\alpha \in \mathbb{R}$. For the sake of simplicity, the calculation will be done asymptotically where sums can be replaced with integrals. Also, the initial cycle is omitted in the analysis because it only adds a constant. Thus, asymptotically, the total number of handles is:

$$k = \sqrt{\frac{2|V|}{\alpha}} \tag{2}$$

which comes from the equation:

$$|V| = \int \alpha i \, di = \alpha \frac{i^2}{2} \tag{3}$$

The estimation of the time consumed by the algorithm over such a handle decomposition is:

$$\int t\left(\alpha i, \alpha \frac{i^2}{2}\right) di \tag{4}$$

In particular, for the *BIBOX* algorithm, the following estimation of time can be obtained as a simple integration over all the handles as follows:

$$\int d_1 \cdot (\alpha i)^3 + d_2 \cdot (\alpha i)^2 \left(\alpha \frac{i^2}{2}\right) + d_3 \cdot (\alpha i)\left(\alpha \frac{i^2}{2}\right)^2 di = \\ = \alpha^3 \left(\frac{1}{4} d_1 \cdot i^4 + \frac{1}{10} d_2 \cdot i^5 + \frac{1}{24} d_3 \cdot i^6\right) \tag{5}$$

If the total number of handles is substituted into (5), the following running time of the *BIBOX* algorithm is obtained:

$$d_1\alpha \cdot |V|^2 + \frac{2}{5}d_2\sqrt{\alpha} \cdot |V|^2\sqrt{|V|} + \frac{1}{3}d_3 \cdot |V|^3 \tag{6}$$

The asymptomatic running time is $\mathcal{O}(|V|^3)$, where the most important addend is $\frac{1}{3}d_3 \cdot |V|^3$. The sum of the remaining addends in the expression will be called a *residuum*. In the case of the *BIBOX* algorithm with linear sizes of handles, the residuum is as follows:

$$d_1\alpha \cdot |V|^2 + \frac{2}{5}d_2\sqrt{\alpha} \cdot |V|^2\sqrt{|V|} \in \mathcal{O}\left(|V|^{2+\frac{1}{2}}\right) \tag{7}$$

Naturally, the smallest possible residuum is desirable for having the shortest possible running time of the algorithm. Fortunately, the residuum can be affected by the handle decomposition – particularly by handle sizes shown in the above calculation.

Proposition 1 ***(residuum in various handle decompositions).*** For standard cases of handle decompositions, where the size of the handle L_i is $\mathcal{O}(1)$, $\mathcal{O}(\sqrt{i})$, $\mathcal{O}(i^2)$, or $\mathcal{O}(2^i)$, respectively, the following estimates of the residuum are obtained:

- if $|L_i| = \beta; \beta \in \mathbb{R} \Rightarrow \mathcal{O}(|V|^2)$
- if $|L_i| = \zeta\sqrt{i}; \zeta \in \mathbb{R} \Rightarrow \mathcal{O}(|V|^{2+\frac{1}{3}})$
- if $|L_i| = \gamma i^2; \gamma \in \mathbb{R} \Rightarrow \mathcal{O}(|V|^{2+\frac{2}{3}})$
- if $|L_i| = \delta 2^i; \delta \in \mathbb{R} \Rightarrow \mathcal{O}(|V|^3)$. ■

Since the calculations are analogous to the linear size of the handle, the proof is deferred to the Appendix.

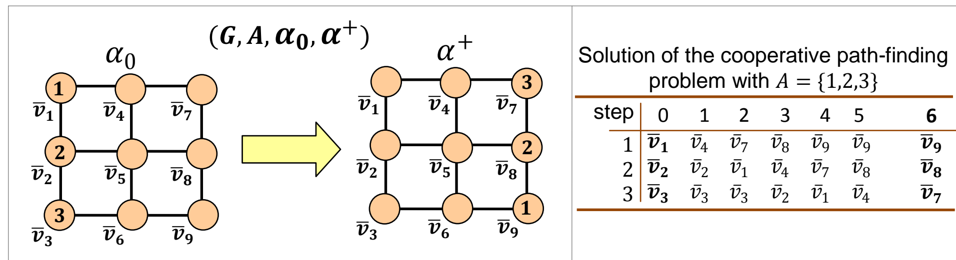
According to the size of the residuum, it seems that, with respect to reducing the runtime, the best handle decomposition is that with a constant handle size while the worst one is that with an exponential handle size.

4. Experimental Evaluation in the Case of Cooperative Path-finding

It is interesting to explore whether theoretical estimations match the real situation. That is, whether a handle decomposition with handles of a constant size really is

the best option for the *BIBOX* algorithm. To provide a complete picture, the problem is introduced in the following definition.

Definition 3 ***(cooperative path-finding)*** [1, 9]**.** Let $G = (V, E)$ be an undirected graph and $A = \{a_1, a_2, \ldots, a_n\}$, where $|A| < |V|$ be a set of agents. The *arrangement* of agents in the graph reflects the uniquely invertible function $\alpha: A \longrightarrow V$ (there is at most one agent in each vertex). The problem of *cooperative path-finding* (CPF) consists in finding a sequence of moves of agents so that the given initial arrangement α_0 is transformed to the given goal arrangement α^+. The move with an agent is possible along an edge. □



step	0	1	2	3	4	5	6
1	$\bar{v}_1$	$\bar{v}_4$	$\bar{v}_7$	$\bar{v}_8$	$\bar{v}_9$	$\bar{v}_9$	$\bar{v}_9$
2	$\bar{v}_2$	$\bar{v}_2$	$\bar{v}_1$	$\bar{v}_4$	$\bar{v}_7$	$\bar{v}_8$	$\bar{v}_8$
3	$\bar{v}_3$	$\bar{v}_3$	$\bar{v}_3$	$\bar{v}_2$	$\bar{v}_1$	$\bar{v}_4$	$\bar{v}_7$

Figure 1. *An example of the cooperative path-finding problem (CPF).* Three agents need to be rearranged in a 3×3 grid. A solution of length 6, where multiple agents move in a single step, is shown.

The *inverse arrangement* of agents is the mapping $\bar{\alpha}: V \longrightarrow A \cup \{\perp\}$ that provides information about which agent is located in a given vertex. The special value $\perp$ indicates that the given vertex is empty.

A number of variants of the problem need to be considered. In this case, a variant with an agent moving into an empty vertex through an edge is concerned (a variant where a sequence of agents can move at once like a train is discussed in [11]). An example instance and its solution are shown in Figure 1.

The *BIBOX* algorithm, for which the analysis has been designed, solves the problem over bi-connected graphs. It arranges agents into handles while the problem is inductively reduced on a smaller bi-connected graph whenever agents are arranged into the handle – agents in such a handle do not move any more.

Instead of measuring runtime, the number of generated moves will be evaluated. The number of generated moves corresponds exactly to the runtime. The reason for using a number of moves instead of runtime is that it is impossible to measure runtime as precisely as the number of moves, furthermore, it is less dependent on the implementation.

Here, the algorithm is recalled using a pseudo-code as Algorithm 1. It employs several auxiliary functions to solve subtasks. The pseudo-code of auxiliary functions is given in [11] – at this point, they are only briefly described.

Algorithm 1. *The **BIBOX** algorithm.* The algorithm solves the cooperative path-finding problem (CPF) over bi-connected graphs consisting of a cycle and at least one handle with two unoccupied vertices. The algorithm proceeds inductively according to a handle decomposition. The two unoccupied vertices are necessary for arranging agents within the initial cycle; in the rest of the graph only one unoccupied vertex is needed. A pseudo-code is built around several higher-level operations:

- Lock(U) — locks all the vertices from set U; each vertex is either *locked* or *unlocked*; an agent must not be moved out of the locked vertex, which is respected by other operations
- Unlock(U) — unlocks all the vertices from set U
- Make-Unoccupied(v) — vacates vertex v
- Move-Agent(a, v) — moves agent a from its current location to vertex v
- Rotate-Cycle$^+$(C) — rotates cycle C in the positive direction; a vacant vertex must be present in the cycle
- Rotate-Cycle$^-$(C) — rotates cycle C in the negative direction
- Transform-Goal(G, A, α^+) — transforms goal arrangement α^+ to a new arrangement so that finally unoccupied vertices are located in the initial cycle of the handle decomposition; two disjoint paths along which empty vertices are relocated are returned
- Finish-Solution(φ, χ) — transforms the arrangement with two unoccupied vertices in the initial cycle to the original goal arrangement; φ and χ are two disjoint paths along which empty vertices shifted
- Solve-Original-Cycle — arranges agents within the initial cycle of the handle decomposition to comply with the transformed goal arrangement; two empty vertices are employed to arrange agents

```
procedure BIBOX-Solve(G = (V, E), A, α_0, α^+)
    /* Top level function of the BIBOX algorithm; solves
    a given cooperative path-finding problem.
    Parameters:   G    – a graph modeling the environment,
                  A    – a set of agents,
                  α_0  – the initial arrangement of agents,
                  α^+  – the goal arrangement of agents. */
1:  let D = [C_0, L_1, L_2, ..., L_k] be a handle decomposition of G
2:  (α^+, φ, χ) ← Transform-Goal(G, A, α^+)
3:  α ← α_0
4:  for c = k, k − 1, ..., 1 do
5:  |   if |L_c| > 2 then
6:  |_  |_  Solve-Regular-Handle(c)
7:  Solve-Original-Cycle
8:  Finish-Solution(φ, χ)
```

```
procedure Solve-Regular-Handle(c)
    /* Places agents the destinations of which are within
    The handle L_c; agents placed in the handle L_c are finally
    locked to prevent them from moving.
    Parameters:     c – index of a handle */
9:  let [u, w_1, w_2, ..., w_l, v] = L_c
    /* Both unoccupied vertices must be located
    outside the currently solved handle. */
10: let x, z ∈ V \ ∪_{c=j}^{k} (L_j \ {u, v}) such that x ≠ z
11: Make-Unoccupied(x)
12: Lock({x})
13: Make-Unoccupied(z)
14: Unlock({x})
15: for i = l, l − 1, ..., 1 do
16:     Lock(L_c \ {u, v})
        /* The agent to be placed is outside the handle L_c. */
17:     if α(ᾱ⁺(w_i)) ∉ (L_c \ {u, v}) then
18:         Move-Agent(ᾱ⁺(w_i), u)
19:         Lock({u})
20:         Make-Unoccupied(v)
21:         Unlock(L_c)
22:         Rotate-Cycle⁺(C(L_c))
        /* The agent to be placed is inside the handle L_c. */
23:     else
24:         Make-Unoccupied(u)
25:         Unlock(L_c)
26:         ρ ← 0
27:         while α(ᾱ⁺(w_i)) ≠ v do
28:             Rotate-Cycle⁺(C(L_c))
29:             ρ ← ρ + 1
30:         Lock(L_c \ {u, v})
31:         let y ∈ V \ (∪_{j=c+1}^{d} (L_j \ {u, v}) ∪ C(L_j))
32:         Move-Agent(ᾱ⁺(w_i), y)
33:         Lock ({y})
34:         Make-Unoccupied(u)
35:         Unlock(L_c)
36:         while ρ > 0 do
37:             Rotate-Cycle⁻(C(L_c))
38:             ρ ← ρ − 1
39:         Unlock({y})
40:         Lock(L_c \ {u, v})
41:         Move-Agent(ᾱ⁺(w_i), u)
42:         Lock ({u})
43:         Make-Unoccupied(v)
44:         Unlock(L_c)
45:         Rotate-Cycle⁺(C(L_c))
46: Lock(L_c \ {u, v})
```

The algorithm starts with the construction of a handle decomposition (line 1). This step is non-deterministic in the original algorithm and can be replaced with a handle decomposition where sizes of handles satisfy certain conditions. It is assumed that a cycle denoted as $C(L_i)$ is associated with each handle; $C(L_i)$ can be constructed by adding a path connecting the handle's connection vertices u and v. Thereafter, the goal arrangement of agents is transformed so that the vacant vertices are eventually located in the initial cycle of the decomposition (line 2). In fact, the algorithm solves this modified instance. The original instance is solved by relocating vacant vertices from the initial cycle to their original goal locations (line 8). This instance transformation is carried out using the auxiliary functions *Transform-Goal*, and *Finish-Solution* that relocate vacant vertices along two vertex disjoint paths. The main loop (lines 4-6) processes a handle from the last one towards the initial cycle. Agents are arranged by means of another auxiliary procedure, *Solve-Original-Cycle*, in the original cycle (line 7).

Individual handles are processed by the *Solve-Regular-Handle* procedure. It arranges agents into a handle in a stack-like manner. First, unoccupied vertices are moved out of the processed handle as they will be needed elsewhere (lines 10-14). Subsequently, agents, whose goal positions are in the handle, are processed. Two cases are distinguished depending on whether the processed agent is located outside the handle (lines 17-22) or within the handle (lines 23-45). The case is with the agent outside is easier to solve – in this case, the agent is moved to the connection vertex u using the *Move-Agent* auxiliary procedure. The other connection vertex v is vacated by the *Make-Unoccupied* procedure. If some vertex is free in the cycle $C(L_c)$ then the cycle can be rotated. This is performed once in the positive direction using the *Rotate-Cycle*$^{+}$ function. The rotation places the agent into the handle. Throughout the agent relocation process, vertex locking is used (functions *Lock* and *Unlock*) to fix the agent in a certain vertex while other agents or the vacant vertex are relocated.

A more difficult situation occurs when the agent is placed inside the handle. In such a case, the agent must be rotated out of the handle to the rest of the graph (lines 24-29). The number of positive rotations to get the agent out of the handle is counted (lines 27-29). The counted number of rotations is used to restore the situation with the corresponding number of negative rotations (lines 36 – 38). At this point, the situation is the same as in the previous case. Thus, the agent is stacked into the handle in the same way.

Consider that n agents need to be arranged into a handle of size n, which is connected to a bi-connected graph of size m. Processing a single agent requires rotating the handle no more than $2n$+1 times – at most n rotations are needed to get the agent out of the handle if it is originally located inside; at most n rotations

are needed to rotate the handle back; and one rotation is required to push the agent inside the handle. Each rotation requires $n + m$ steps as, in the worst case, the cycle that is rotated can include the whole graph. The time required for the rotations can be thus estimated by $d \cdot (2n + 1) \cdot (n + m)$ with some $d \in \mathbb{R}$. Hence, the time needed for the rotations when processing a single handle can be estimated by $d_1 \cdot n^3 + d_2 \cdot n^2 m$ $d_1, d_2 \in \mathbb{R}$. To finish the estimation, it is needed to account for the time needed to relocate the agent towards the handle's connection vertex from inside the graph or between the handle's connection vertices. Both cases can be estimated using $d_3 \cdot m^2$ since the agent needs to be moved along a path of a length of at most m, where traversing a single edge requires at most m steps (a destination vertex must be vacated each time the edge is traversed). Altogether, the expression $t(n, m) = d_1 \cdot n^3 + d_2 \cdot n^2 m + d_3 \cdot nm^2$, where $d_1, d_2, d_3 \in \mathbb{R}$ estimates the time needed for processing the handle. The argumentation is based on the fact that performing a move with an agent corresponds to a constant time.

4.1. *Measurement of the Number of Moves*

An experimental evaluation has been made to check if the theoretical analysis matches the real-life situation. The number of moves generated by the *BIBOX* algorithm with different sizes of handles was measured. All the theoretically studied cases of the size of the handle L_i $\mathcal{O}(1)$, $\mathcal{O}(i)$, $\mathcal{O}(\sqrt{i})$, $\mathcal{O}(i^2)$ and $\mathcal{O}(2^i)$ were tested. The multiplication factor of 1 was used in all the cases to generate a sequence of handles of various sizes; that is, $\alpha, \beta, \gamma, \delta, \zeta = 1$. Ten instances of the CPF problem were generated for each size of the constructed bi-connected graph. Random initial and goal arrangements with exactly two unoccupied vertices were generated in each instance. The results are shown in Figure 2.

The experimental evaluation clearly matches the derived theoretical results; that is, constant size handles produce the lowest number of moves, while handles of the exponential size produce the most moves. The relative ordering of the number of moves in other handle sizes in the experimental evaluation is the same as in the theoretical analysis. It seems that differences in the number of moves in the experimental evaluation are even more pronounced than in the theoretical analysis. The explanation is that a graph with smaller handles exhibits higher connectivity that implies shorter paths along which agents are relocated.

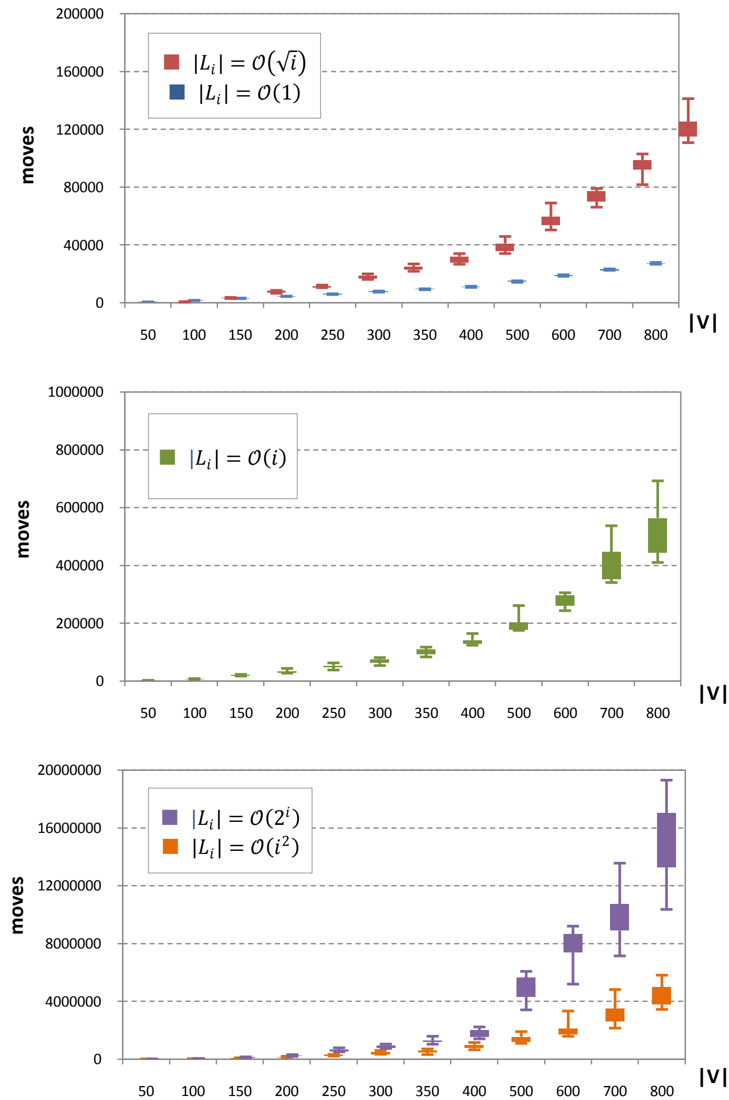


Figure 2. *Number of moves generated by the BIBOX algorithm.* The algorithm was tested on bi-connected graphs with the size of the i-th handle corresponding to the *constant*, $\sqrt{i}$, i, i^2, and 2^i respectively. The lowest number of moves is obtained for the constant handle size, and the highest number for handles, the size of which is exponential.

5. Conclusions and Future Work

A simple but general framework for evaluating induction-based algorithms for processing bi-connected graphs has been introduced. It is assumed that algorithms process bi-connected graphs according to their decomposition into handles. The last handle is processed first, which reduces the processing task on a smaller bi-connected graph – the graph from which the last handle has been removed. The impact of the handle decomposition on the performance of the algorithm was studied.

A theoretical and experimental evaluation of a particular case of an algorithm solving the cooperative path-finding (CPF) problem over bi-connected graphs was performed. The theoretical evaluation showed that among several suggested handle decompositions the decomposition with the constant size of handles is the best option with respect to the number of moves solving the given CPF instance. The performed experimental evaluation confirmed the theoretical assumptions.

The described framework identifies the best decomposition among several known decompositions. It is worth further exploration whether a general mechanism for determining the best handle decomposition can be found.

References

1. **Cheriton**, D., **Tarjan**, R. E.: *Finding Minimum Spanning Trees.* SIAM Journal on Computing, Volume 5, pp. 724–741, Society for Industrial and Applied Mathematics, 1976.
2. **Dijkstra**, E. W.: *A Note on Two Problems in Connection with Graphs.* Numerische Mathematik, Volume 1, pp. 269–271, Springer Verlag, 1959.
3. **Jarník**, V.: *O jistém problému minimálním (About a Certain Minimal Problem).* Práce Moravské Přírodovědecké Společnosti, Volume 6, pp. 57–63, Moravská Přírodovědecká Společnost, 1930.
4. **Kornhauser**, D., **Miller**, G. L., and **Spirakis**, P. G.: *Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications.* Proceedings of the 25th Annual Symposium on Foundations of Computer Science (FOCS 1984), pp. 241-250, IEEE Press, 1984.
5. **Kruskal**, J. B.: *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem.* Proceedings of the American Mathematical Society, Volume 7, pp. 48–50, American Mathematical Society, 1956.
6. **Nešetřil**, J., **Milková**, E., **Nešetřilová**, H. *Otakar Borůvka on Minimum Spanning Tree Problem: Translation of both the 1926 Papers, Comments, History*. Discrete Mathematics, Volume 233 (1–3), pp. 3–36, Elsevier, 2001.

7. **Prim**, R. C.: *Shortest Connection Networks and Some Generalizations.* Bell System Technical Journal, Volume 36, pp. 1389–1401, American Telephone and Telegraph Company, 1957.
8. **Ryan**, M. R. K.: *Exploiting Subgraph Structure in Multi-Robot Path Planning*, Journal of Artificial Intelligence Research (JAIR), Volume 31, pp. 497-542, AAA Press, 2008.
9. **Silver**, D.: *Cooperative Pathfinding.* Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2005), pp. 117-122, AAAI Press.
10. **Surynek**, P.: *A Novel Approach to Path Planning for Multiple Robots in Bi-connected Graphs.* Proceedings of the 2009 IEEE International Conference on Robotics and Automation (ICRA 2009), pp. 3613-3619, IEEE Press, 2009.
11. **Surynek**, P.: *Solving Abstract Cooperative Path-Finding in Densely Populated Environments.* Computational Intelligence (COIN), Volume 30, Issue 2, pp. 402-450, Wiley, 2014.
12. **Tarjan**, R. E.: *Depth-First Search and Linear Graph Algorithms.* SIAM Journal on Computing, Volume 1 (2), pp. 146-160, Society for Industrial and Applied Mathematics, 1972.
13. **Westbrook**, J., **Tarjan**, R. E.: *Maintaining Bridge-connected and Biconnected Components On-line.* Algorithmica, Volume 7, Number 5&6, pp. 433–464, Springer Verlag, 1992.
14. **de Wilde**, B., **ter Mors**, A., **Witteveen**, C.: *Push and Rotate: Cooperative Multi-agent Path Planning.* Proceedings of International conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2013), IFAAMAS, 2013, pp. 87-94.

Appendix

Proof of **Proposition 1:** Assume that $|L_i| = \beta$; $\beta \in \mathbb{N}$; that is $|L_i| = \mathcal{O}(1)$. The number of handles, which will be used as a bound in the integration, is:

$$k = \frac{|V|}{\beta}$$

which can be obtained from the equation:

$$|V| = \int \beta \, di = \beta i$$

The number of moves generated by the algorithm in such a case can be obtained from the following integration:

$$\int d_1 \cdot \beta^3 + d_2 \cdot \beta^2(\beta i) + d_3 \cdot \beta(\beta i)^2 di =$$
$$= \beta^3 \left(d_1 \cdot i + \frac{1}{2} d_2 \cdot i^2 + \frac{1}{3} d_3 \cdot i^3\right)$$

After substituting k into the expression, the following number of moves is obtained:

$$d_1\beta^2 \cdot |V| + \frac{1}{2} d_2\beta \cdot |V|^2 + \frac{1}{3} d_3 \cdot |V|^3$$

Hence, the residuum is:

$$d_1\beta^2 \cdot |V| + \frac{1}{2} d_2\beta \cdot |V|^2 \in \mathcal{O}(|V|^2)$$

Assume that $|L_i| = \gamma i^2$; $\gamma \in \mathbb{N}$; that is $|L_i| = \mathcal{O}(i^2)$. The number of handles, which will be used as a bound in the integration, is:

$$k = \sqrt[3]{\frac{3|V|}{\gamma}}$$

which can be obtained from the equation:

$$|V| = \int \gamma i^2 \, di = \frac{1}{3}\gamma i^3$$

In such a case, the number of moves generated by the algorithm can be obtained from the following integration:

$$\int d_1 \cdot (\gamma i^2)^3 + d_2 \cdot (\gamma i^2)^2 \left(\frac{1}{3}\gamma i^3\right) + d_3 \cdot \gamma i^2 \left(\frac{1}{3}\gamma i^3\right)^2 di =$$
$$= \gamma^3 \left(\frac{1}{7} d_1 \cdot i^7 + \frac{1}{24} d_2 \cdot i^8 + \frac{1}{81} d_3 \cdot i^8\right)$$

After substituting k into the expression, the following number of moves is obtained:

$$\frac{9\sqrt[3]{3}}{7} d_1\left(\sqrt[3]{\gamma}\right)^2 \cdot |V|^2 \sqrt[3]{|V|} + \frac{3(\sqrt[3]{3})^2}{8} d_2 \sqrt[3]{\gamma} \cdot |V|^2 (\sqrt[3]{|V|})^2 + \frac{1}{3} d_3 \cdot |V|^3$$

Hence, the residuum is:

$$\frac{9\sqrt[3]{3}}{7}d_1\left(\sqrt[3]{\gamma}\right)^2 \cdot |V|^2\sqrt[3]{|V|} + \frac{3(\sqrt[3]{3})^2}{8}d_2\sqrt[3]{\gamma} \cdot |V|^2(\sqrt[3]{|V|})^2 \in \mathcal{O}(|V|^{2+\frac{2}{3}})$$

Assume that $|L_i| = \delta 2^i$; $\delta \in \mathbb{R}$; that is, $|L_i| = \mathcal{O}(2^i)$. The number of handles, which will be used as a bound in the integration, is:

$$k = \log_2 |V| - \log_2 \left(\frac{\delta}{\ln 2}\right)$$

which can be obtained from the equation:

$$|V| = \int \delta 2^i \, di = \frac{\delta}{\ln 2} 2^i$$

The number of moves generated by the algorithm can be obtained from the following integration:

$$\int \left(d_1 + \frac{d_2}{\ln 2} + \frac{d_3}{\ln^2 2}\right) \cdot (\delta 2^i)^3 di =$$
$$= \frac{1}{3}\left(\frac{d_1}{\ln 2} + \frac{d_2}{\ln^2 2} + \frac{d_3}{\ln^3 2}\right)\delta^3 \cdot 2^{3i}$$

After substituting k into the expression, the following number of moves is obtained:

$$\frac{1}{3}(d_1 \ln^2 2 + d_2 \ln 2 + d_3) \cdot |V|^3$$

Hence, the residuum is:

$$\frac{1}{3}(d_1 \ln^2 2 + d_2 \ln 2) \cdot |V|^3 \in \mathcal{O}(|V|^3)$$

Assume that $|L_i| = \zeta\sqrt{i}$; $\zeta \in \mathbb{R}$; that is, $|L_i| = \mathcal{O}(\sqrt{i})$. The number of handles, which will be used as a bound in the integration, is:

$$k = \sqrt[3]{\frac{9}{4}\left(\frac{|V|}{\zeta}\right)^2}$$

which can be obtained from the equation:

$$|V| = \int \zeta \sqrt{i} di = \zeta \frac{2}{3} i^{\frac{3}{2}}$$

The number of moves generated by the algorithm can be obtained from the following integration:

$$\int d_1 \cdot (\zeta\sqrt{i})^3 + d_2 \cdot \left(\zeta\sqrt{i}\right)^2 \left(\zeta \frac{2}{3} i^{\frac{3}{2}}\right) + d_3 \cdot \left(\zeta\sqrt{i}\right)\left(\zeta \frac{2}{3} i^{\frac{3}{2}}\right)^2 di =$$
$$= \zeta^3 \left(\frac{2}{5} d_1 \cdot i^{\frac{5}{2}} + \frac{4}{21} d_2 \cdot i^{\frac{7}{2}} + \frac{1}{9} d_3 \cdot i^{\frac{9}{2}}\right)$$

After substituting k into the expression, the following number of moves is obtained:

$$\left(\frac{3}{2}\right)^{\frac{5}{3}} d_1 \zeta^{\frac{4}{3}} \cdot |V|^{\frac{5}{3}} + \frac{4}{21}\left(\frac{3}{2}\right)^{\frac{7}{3}} d_2 \zeta^{\frac{17}{6}} \cdot |V|^{\frac{7}{3}} + \frac{1}{3} d_3 \cdot |V|^3$$

Hence, the residuum is:

$$\left(\frac{3}{2}\right)^{\frac{5}{3}} d_1 \zeta^{\frac{4}{3}} \cdot |V|^{\frac{5}{3}} + \frac{4}{21}\left(\frac{3}{2}\right)^{\frac{7}{3}} d_2 \zeta^{\frac{17}{6}} \cdot |V|^{\frac{7}{3}} \in \mathcal{O}(|V|^{2+\frac{1}{3}})$$

■

Pavel **Surynek**: *Redundancy Elimination in Highly Parallel Solutions of Motion Coordination Problems.* International Journal on Artificial Intelligence Tools (IJAIT), Volume 22, Number 05 (19 pages), World Scientific, 2013, ISSN 0218-2130.

Redundancy Elimination in Highly Parallel Solutions of Motion Coordination Problems

Pavel Surynek

Charles University Prague
Faculty of Mathematics and Physics
Department of Theoretical Computer Science and Mathematical Logic
Malostranské náměstí 25, Praha, 118 00, Czech Republic
pavel.surynek@mff.cuni.cz

Abstract. Problems of motion coordination of multiple entities are addressed in this paper. These problems are dealt on the abstract level where they can be viewed as tasks of constructing a spatial-temporal plan for a set of identical mobile entities. The entities reside in a certain environment where they can move. Each entity need to reach a given goal position supposed it is starting from some initial position. The most abstract formal representations of coordinated motion problems are known as "pebble motion on a graph" and "multi-robot path planning". The existent algorithms for pebble motion and multi-robot problems were suspected of generating solutions containing redundancies. This hypothesis eventually confirmed in this work. We present several techniques for identifying and eliminating redundancies from solutions generated by existent algorithms. An extensive experimental evaluation was performed and it showed that the quality of generated solutions can be improved up to the order of magnitude. We also identify parameters characterizing instances of problems where a significant improvement is expectable.

***Keywords*:** multi-robot path planning; pebble motion on a graph; redundancy elimination; parallel plans; SAT based optimization.

1. Introduction, Context, and Motivation

Problems of coordinated motion of multiple identical entities as they are introduced in [4, 8, 10, 16] (terms "multi-robot path planning" or "cooperative path-

finding" are also used to denote the same or similar problem) represent a basic abstraction for many real-life and theoretical tasks. The classical task that can be abstracted as a problem of coordinated motion takes place in a certain physical environment where identical mobile entities are moving (typically represented by mobile robots). Each entity is given its initial and goal positions in the environment between which it should relocate. The task is to construct a spatial-temporal plan for all the entities such that they can reach their goal positions following the plan while the plan satisfies certain natural constraints. These constraints are constituted by a requirement that the entities must avoid obstacles in the environment and must not collide with each other.

The standard abstraction adopted throughout this work uses an undirected graph to model the environment. Vertices of this graph represent positions in the environment and edges represent passable regions between two positions. An arrangement of entities in the environment is abstracted as a uniquely invertible assignment of entities to vertices. At least one vertex remains unoccupied in order to make the movement of entities possible – for example moving in a circle where each entity follows the preceding entity is not allowed. The time is discrete; it is an ordered set of time steps isomorphic to the structure of natural numbers. A way in which an arrangement of entities can be transformed into another can slightly differ in variants of the problem. The best known abstract formalizations of coordinated motion problems are represented by *pebble motion on a graph* (PMG) as defined in [4] and [16], and *multi-robot path planning* (MRPP) as defined in [8, 10, 11] while the latter allows higher parallelism.

Abstract problems of coordinated motion of multiple entities on a graph are motivated by many real-life problems. The most typical motivating example is motion planning of a group of mobile robots that are moving in 2-dimensional space. Generally, if there is enough free space in the environment, algorithms based on search for shortest paths in a graph with an eventual local repairs if collision occurs can be used [1]. However, if non-trivial amount of space is occupied different approaches must be adopted.

Many well known puzzles can be formulated as coordinated motion on a graph. The best known is so called Lloyd's 15-puzzle and its generalizations as described in [6, 7] and [16]. In practice, entities may be represented by various mobile or movable objects – for example rearranging containers in some storage area can be interpreted as a problem of coordinated motion where entities are represented by containers. Exactly this interpretation has been used for planning motions of automated straddle carriers in a storage area in Patrick port facility at Port Brisbane in Queensland as reported in [8]. Although the approach suggested in [8] is applied on few movable entities it clearly demonstrates the usefulness of

discussed abstractions. Entities do not necessarily need to be physical objects. Virtual spaces of computer simulations and games contain many situations where motions of certain entities must be planned. A typical example is a coordination of groups of units in real-time strategic computer games (RTS) [14].

It is necessary to emphasize that contrary to multi-agent motion planning [3], the centralized approach is adopted in this work. This means that the environment is fully observable for the central planning mechanism and the individual entities merely execute the centrally created plan.

There exist several relatively efficient methods for solving problems of coordinated motion on a graph. This work is particularly targeted on solving methods described in [10, 11]. These methods represent algorithms for the class of problems where the graph modeling the environment is bi-connected [15] and the graph is densely occupied by entities. More precisely, the number of entities μ is comparable to the size of the set of vertices (that is, $\mu = \Theta(|V|)$). Despite the good performance of these methods, generated solutions are suspected of containing certain redundancies. This is a hypothesis whose examination is the main contribution of this paper. If it is the case that generated solutions contain redundancies, then a question how they can be removed to improve the solution arises.

The task was to analyze solutions of non-trivial size, which turned out to be infeasible to be done manually. Moreover, we were searching redundancies of a priori unknown nature. Therefore, a software tool GraphRec [5] allowing visual analysis of solutions of problems of motion on a graph has been developed and employed in this analysis. Several types of redundancies were observed using the GraphRec software in generated solutions. The most prominent three of them that we manage to formally capture are described in this paper. Methods for automated discovering and elimination of these three defined types of redundancies are suggested and analyzed theoretically as well as experimentally. We also suggest to model the problem of motion on a graph as propositional satisfiability (SAT) [1] which allows us to discover very generic redundancies automatically.

The top level organization of the paper has two parts. The first part explains a specific variant of the coordinated motion problem (section 2) and the basic solving algorithm (section 3); this part mostly recalls existing concepts. The second part contains the main contribution of this work; redundancy elimination methods are described (section 4), and the benefit of suggested methods is justified in the experimental section (section 5). Additionally a SAT based solution improvement technique is described in section 6.

2. Pebble Motion on a Graph (PMG)

In the rest of the paper, we restrict ourselves on the variant of the entity motion coordination problem known as *pebble motion on a graph* (PMG) defined in [7] and [16]. The work can be extended on other variants of the problem such as *multi-robot path planning* (MRPP) using minor modifications only.

The task in pebble motion on a graph is given by an undirected graph with an *initial* and a *goal arrangement* of pebbles in vertices of the graph. Each vertex contains at most one pebble (which represents a movable entity) and at least one vertex remains unoccupied. The task is to find a sequence of moves for each pebble such that all the pebbles reach their goal vertices. A pebble can move into a neighboring unoccupied vertex while no other pebble is entering the same target vertex simultaneously. The following definition formalizes the problem. An illustration of the problem is shown in Fig. 1.

Definition 1 ***(pebble motion on a graph).*** Let $G = (V, E)$ be an undirected graph and let $P = \{p_1, p, \ldots, p_\mu\}$ be a set of pebbles where $\mu < |V|$. The initial arrangement of pebbles is defined by an injective function $S_P^0 : P \longrightarrow V$ (that is $S_P^0(p_i) \neq S_P^0(p_j)$ for $i, j = 1,2, \ldots, \mu$ with $i \neq j$); the goal arrangement of pebbles is defined by another injective function $S_P^+ : P \longrightarrow V$. A problem of PMG is the task to find a number ξ and a sequence $S_P = [S_P^0, S_P^1, \ldots, S_P^\xi]$ where $S_P^k : P \longrightarrow V$ is an injective function for every $k = 1,2, \ldots, \xi$. The following constraints must hold for S_P:

(i) $S_R^\zeta = S_R^+$, that is, pebbles eventually reach their destinations.

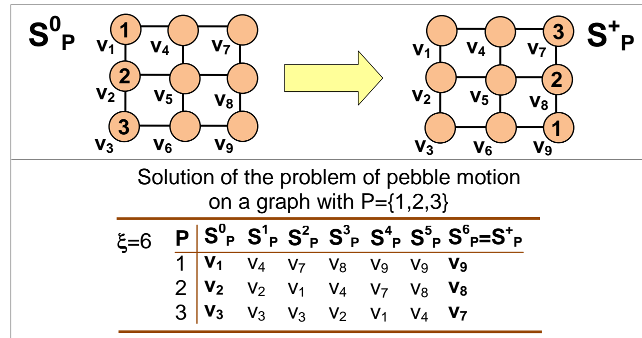
(ii) Either $S_P^k(p) = S_P^{k+1}(p)$ or $\{S_P^k(p), S_P^{k+1}(p)\} \in E$ for every $p \in P$ and $k = 1,2, \ldots, \xi - 1$.

(iii) $S_P^k(p) \neq S_P^{k+1}(p)$ and $S_P^k(q) \neq S_P^{k+1}(q)$ for $\forall q \in P$ such that $q \neq p$ must hold for every $p \in P$ and $k = 1,2, \ldots, \xi - 1$, that is no two pebbles can enter the same target vertex simultaneously.

The problem described above is formally a quadruple $\Pi = (G = (V, E), P, S_P^0, S_P^+)$. □

In practice, the *quality of solution* matters. The typical measures of the quality of solution are its length (the total number of moves) and the makespan (which corresponds to the number ξ). These numbers are required to be small. Unfortunately, requiring either the length of solution or its makespan to be as small as possible makes the problem intractable [7] (the decision variant of the problem is *NP*-complete). On the other hand, if there is no requirement on the quality, the question whether there exists a solution is in the *P* class as it shown in [4] and [16].

However, methods showing evidence that the problem belongs to the P class described in [4] and [16] generates excessively long solutions that are unsuitable for practice when each movement of an entity represented by a pebble has a non-trivial cost. Therefore, it was necessary to find a compromise between the quality of solution and computational effort of its construction. Methods following this compromise are described in [10] and [11]. Solutions produced by these methods were submitted for analysis into the visualization tool in order to find if and how they can be further improved.



P	S^0_P	S^1_P	S^2_P	S^3_P	S^4_P	S^5_P	$S^6_P=S^+_P$
1	v_1	v_4	v_7	v_8	v_9	v_9	v_9
2	v_2	v_2	v_1	v_4	v_7	v_8	v_8
3	v_3	v_3	v_3	v_2	v_1	v_4	v_7

Fig 1. *An illustration of a **PMG problem***. The task is to move pebbles from their initial positions specified by S_P^0 to the goal positions specified by S_P^+. A solution of length 6 is shown.

3. Solving Coordinated Motion Problems

This section is devoted to a brief recall of algorithms described in [10] and [11]. Understanding how these algorithms work will provide us an insight into the structure of solutions produced by them. This theoretical insight founded the hypothesis that solutions can be further improved.

A very important class of pebble motion problems is formed by those whose graph is *bi-connected* which intuitively means that each pair of vertices is connected by two disjoint paths.

Definition 2 *(connectivity, bi-connectivity).* An undirected graph $G = (V, E)$ is *connected* if $|V| \geq 2$ and for every pair of distinct vertices $u, v \in V$ there exists a path connecting u and v in G. An undirected graph $G = (V, E)$ is *bi-connected* if $|V| \geq 3$ and for every vertex $u \in V$ the graph $G' = (V - \{u\}, E \cap \{\{v, w\} | v, w \in V \wedge v \neq u \wedge w \neq u\})$ is connected. □

The importance of this class of problems is assessed by the fact that they are almost always solvable. Moreover, spatial environments in real tasks are often abstracted as two dimensional grids which are bi-connected in most cases.

If the bi-connected graph contains at least two unoccupied vertices and it is not a cycle, then every goal arrangement of pebbles is reachable from every initial arrangement [10]. If the graph contains just one unoccupied vertex which can be without loss of generality fixed, then any arrangement of pebbles can be regarded as a *permutation* with respect to the initial arrangement.

A permutation is *even* if it can be composed of the even number of transpositions; otherwise it is *odd*. If the goal arrangement represents an even permutation, then the problem is always solvable. In case of an odd permutation, the problem is solvable if and only if the graph contains a cycle of the odd length [16].

An inductive construction of bi-connected graphs by adding *handles* is a pivotal concept in developing solving algorithms. Let $G = (V, E)$ be a graph, a *handle* with respect to G is a sequence of vertices $L = [u, x_1, x_2, \ldots, x_l, v]$, where $u, v \in V$ and $x_i \notin V$ for $i = 1,2, \ldots, l$ (it allowed that $l = 0$). The result of an *addition* of handle L to graph G is a new graph $G' = (V', E')$, where $V' = V \cup \{x_1, x_2, \ldots, x_l\}$ and either $E' = E \cup \{\{u, v\}\}$ if $l = 0$ or $E' = E \cup \{\{u, x_1\}, \{x_1, x_2\}, \ldots, \{x_{l-1}, x_l\}, \{x_l, v\}\}$ if $l \geq 1$. Every bi-connected graph $G = (V, E)$ can be constructed from a cycle by a sequence of handle additions.

3.1. *The BIBOX-θ Solving Algorithm*

The *BIBOX-θ* algorithm [11] solves a case of the PMG problem when the graph is bi-connected and there is single unoccupied vertex. The algorithm provides a good performance for the described class of problems in terms of speed and quality of generated solutions. This is the main reason why solutions produced by this algorithm are studied.

In the first phase, a handle decomposition is found; that is, a cycle - called *initial cycle* - and a sequence of handles is determined. Without loss of generality it is required that the unoccupied vertex within the goal arrangement of pebbles is located in the initial cycle. The algorithm then proceeds inductively according to the handle decomposition from the last handle to the initial cycle with the first handle.

Two properties of bi-connected graphs with at least one unoccupied vertex are exploited while pebbles are placed within handles: (a) every vertex can be made unoccupied (this is even true for a connected graph), (b) every pebble can be moved to an arbitrary vertex. A handle is processed in the following way. An orientation of the handle is chosen first – this orientation determines ordering of

vertices within the handle. The first and the last vertex of the handle are the connection points to the remainder graph.

Then pebbles starting with the pebble whose goal position is in the second vertex of the handle are placed into the handle in the stack manner. The current pebble is moved to the last vertex of the handle.

Two cases are distinguished here. If the pebble is already somewhere in the handle it must be moved outside first. If the current pebble is outside the handle, then it can be moved into the last vertex of the handle using property (b).

After placing the pebble into the last vertex of the handle, the handle is rotated once in the direction to the first vertex. When all the pebbles within the handle are processed the task is to solve the problem of the same type on a smaller graph.

Nevertheless, the stack manner of placing pebbles cannot be applied for the initial cycle and the first handle of the decomposition. The algorithm uses a database containing pre-calculated optimal solutions for transpositions and rotation of pebbles along 3-cycles in graphs consisting of a cycle and a handle. A solution to any solvable instance on the initial cycle with the first handle is then composed of solutions from such a database.

3.2. *A Case with More Unoccupied Vertices*

If there are exactly two unoccupied vertices in the graph an alternative more efficient placing of pebbles in the initial cycle and the first handle can be used [10]. If there are more than two unoccupied vertices in the graph the approach proposed in [11] is to fill all the remaining unoccupied vertices except two with extra pebbles. The instance is then solved by the *BIBOX-θ* algorithm and the solution is post-processed by removing movements of extra pebbles out of the solution.

This approach is however suspected of generating unnecessary movements for original pebbles. Notice that original pebbles have to make quite complicated movements when an extra pebble is being placed into a handle. All these movements of the original pebbles are redundant in fact since movements of the extra pebble will be eventually filtered out.

4. Elimination of Redundancies

Several types of redundancies were discovered using the *GraphRec* software. A formal description of these redundancies and algorithms for their elimination are provided in the following sections. When reasoning about redundancies, it is convenient to assume solutions to be sequential; that is, a solution has just one movement between consecutive time steps. Fortunately, the *BIBOX-θ* algorithm

can produce solutions in this form. A solution of this form can be viewed as a sequence of moves.

The notation $k_i: u_i \rightarrow v_i$ will denote a move of a pebble k_i from a vertex u_i to a vertex v_i commenced at time step i. The move is called *non-trivial* if $u_i \neq v_i$. From the formal point of view, the sequential solution is a sequence of non-trivial moves $\Phi = [k_i: u_i \rightarrow v_i | i = 1,2, \dots, \xi - 1]$ (consistency with Definition 1 is also assumed).

Definition 3 ***(inverse moves).*** Two consecutive moves $k_i: u_i \rightarrow v_i$ and $k_{i+1}: u_{i+1} \rightarrow v_{i+1}$ with $i \in \{1,2, \dots, \xi - 2\}$ are called *inverse* if $k_i = k_{i+1}$, $u_i = v_{i+1}$, and $v_i = u_{i+1}$. □

Observe that a pair of inverse moves can be left out of the solution without affecting its *validity* – resulting sequence still solves the problem. However, elimination of an inverse pair may cause that another pair of inverse moves arises. Hence, it is necessary to remove inverse moves from the solution repeatedly until there are any.

Algorithm 1. ***Elimination of inverse moves.***

function *Erase-Inverse-Moves* (Φ): **sequence**
1: **do**
2: $\eta \leftarrow \emptyset$
3: **let** $[k_1: u_1 \rightarrow v_1, k_2: u_2 \rightarrow v_2, \dots, k_{\xi-1}: u_{\xi-1} \rightarrow v_{\xi-1}] = \Phi$
4: **for** $i = 1,2, \dots, \xi - 1$ **do**
5: **if** $k_i: u_i \rightarrow v_i$ **and** $k_{i+1}: u_{i+1} \rightarrow v_{i+1}$ are inverse **then**
6: $\eta \leftarrow \eta \cup \{k_i: u_i \rightarrow v_i, k_{i+1}: u_{i+1} \rightarrow v_{i+1}\}$
7: $\Phi \leftarrow \Phi - \eta$
8: **while** $\eta \neq \emptyset$
9: **return** Φ

The process of elimination inverse moves is expressed as Algorithm 1. The worst case time complexity of the algorithm is $O(|\Phi|^2)$, the space complexity is $O(|\Phi|)$.

Definition 4 ***(redundant moves).*** A sequence of moves $[k_{i_j}: u_{i_j} \rightarrow v_{i_j} | j = 1,2, \dots, l]$, where $I = [i_j \in \{1,2, \dots, \xi - 2\} | j = 1,2, \dots, l]$ is a an increasing sequence of indices, is called *redundant* if $|\{k_{i_j} | j = 1,2, \dots, l\}| = 1$, $u_{i_1} = v_{i_l}$, and for each move $k_\iota: u_\iota \rightarrow v_\iota$ with $i_1 < \iota < i_l \wedge \iota \notin I$ it holds that $k_\iota \neq k_{i_1} \Rightarrow u_{i_1} \notin \{u_\iota, v_\iota\}$. □

Redundant moves represents generalization of inverse moves (a pair of inverse moves form a redundant sequence). It is a sequence of moves, which relocates a

pebble into some vertex for the second time while the other pebbles do not enter this vertex at any time step between the beginning and the end of the sequence. Eliminating a redundant sequence of moves preserves validity of the solution.

Again, it is necessary to remove redundant sequences repeatedly since its removal may cause that another redundant sequence arises.

Algorithm 2 formalizes the process of removing redundant moves in the pseudo-code. The worst case time complexity is $O(|\Phi|^4)$, the space complexity is $O(|\Phi|)$.

Definition 5 *(long sequence).* Let S_P^t be a set of vertices occupied by pebbles at time step t. A sequence of moves $[k_{i_j}: u_{i_j} \rightarrow v_{i_j} | j = 1,2, \ldots, l]$, where $I = [i_j \in \{1,2, \ldots, \xi - 2\} | j = 1,2, \ldots, l]$ is an increasing sequence of indices, is called *long* if $|\{k_{i_j} | j = 1,2, \ldots, l\}| = 1$ and there exists a path $C = [c_1 = u_{i_1}, c_2, \ldots, c_n = v_{i_l}]$ in G such that $n < l$, $C \cap S_P^{i_1} = \emptyset$, and for all the moves $k_\iota: u_\iota \rightarrow v_\iota$ with $i_1 < \iota < i_l \wedge \iota \notin I$ it holds that $k_\iota \neq k_{i_1} \Rightarrow \{u_\iota, v_\iota\} \cap C = \emptyset$. □

Algorithm 2. *Elimination of redundant moves.*

function *Erase-Redundant-Moves* (Φ): **sequence**
1: **do**
2: $\eta \leftarrow$*Find-Redundant-Moves*(Φ)
3: $\Phi \leftarrow \Phi - \eta$
4: **while** $\eta \neq \emptyset$
5: **return** Φ

function *Find-Redundant-Moves* (Φ): **sequence**
6: **let** $[k_1: u_1 \rightarrow v_1, \ldots, k_{\xi-1}: u_{\xi-1} \rightarrow v_{\xi-1}] = \Phi$
7: **for** $i = 1,2, \ldots, \xi - 2$ **do** {beginning of redundant sequence}
8: **for** $j = \xi - 1, \xi - 2, \ldots, i + 1$ **do** {end of redundant sequence}
9: **if** $k_i = k_j \wedge u_i = v_j$ **then**
10: $\eta \leftarrow \emptyset$ {redundant sequence}
11: **for** $\tau = i, i + 1, \ldots, j$ **do**
12: **if** $k_i = k_\tau$ **then** $\eta \leftarrow \eta \cup \{k_\tau: u_\tau \rightarrow v_\tau\}$
13: **if** *Check-Redundant-Moves*(Φ, i, j) **then return** η
14: **return** $\emptyset$

function *Check-Redundant-Moves* (Φ, i, j): **boolean**
15: **let** $[k_1: u_1 \rightarrow v_1, \ldots, k_{\xi-1}: u_{\xi-1} \rightarrow v_{\xi-1}] = \Phi$
16: **for** $\iota = i + 1, i + 2, \ldots, j - 1$ **do**
17: **if** $k_\iota \neq k_i \wedge u_i \in \{u_\iota, v_\iota\}$ **then return** $False$
18: **return** $True$

The concept of long sequence is a generalization of redundant sequence (the path C is empty in the case of redundant sequence). Intuitively, the long sequence can be replaced by a sequence of moves along a shorter path (cutoff path) into

which other pebbles do not enter between the beginning and the end of the sequence. Replacing a long sequence of moves by a sequence of moves along the path C again preserves validity of the solution. Again, the replacement of long sequences must be performed repeatedly since new long sequences may arise.

The process of replacement is formally expressed as Algorithm 3. The worst case time complexity is $O(|\Phi|^4 + |\Phi|^3|V|^2)$; the space complexity is $O(|\Phi| + |V| + |E|)$.

Redundant moves and long sequences were described manually using the *GraphRec* software. Without the visualization software we would be unable to discover them.

Algorithm 3. Replacement of long sequences.

function *Replace-Long-Moves* (Φ, G): **sequence**
1: **do**
2: $(\eta, \pi) \leftarrow$*FindLongMoves*(Φ, G)
3: $\Phi \leftarrow \Phi - \eta$; $\Phi \leftarrow \Phi \cup \pi$
4: **while** $(\eta, \pi) \neq (\emptyset, [])$
5: **return** Φ

function *Find-Long-Moves* (Φ, G): **pair**
6: **let** $[k_1: u_1 \rightarrow v_1, \dots, k_{\xi-1}: u_{\xi-1} \rightarrow v_{\xi-1}] = \Phi$
7: **for** $i = 1,2,\dots,\xi - 2$ **do**
8: **for** $j = \xi - 1, \xi - 2, \dots, i + 1$ **do**
9: **if** $k_i = k_j$ **then**
10: $\eta \leftarrow \emptyset$
11: **for** $\tau = i, i+1, \dots, j$ **do**
12: **if** $k_i = k_\tau$ **then** $\eta \leftarrow \eta \cup \{k_\tau: u_\tau \rightarrow v_\tau\}$
13: $C \leftarrow$*Check-Long-Moves*$(\Phi, i, j, |\eta|, G)$
14: **if** $C \neq []$ **then**
15: **let** $[c_1, c_2, \dots, c_n] = C$
16: $\pi \leftarrow [k_i: c_1 \rightarrow c_2, \dots, k_i: c_{n-1} \rightarrow c_n]$
17: **return** (η, π)
18: **return** $(\emptyset, [])$

function *Check-Long-Moves* $(\Phi, i, j, l, G = (V, E))$: **sequence**
19: **let** $[k_1: u_1 \rightarrow v_1, \dots, k_{\xi-1}: u_{\xi-1} \rightarrow v_{\xi-1}] = \Phi$
20: $(V', E') \leftarrow G$; $V' \leftarrow V' - S_P^i$; $E' \leftarrow E' \cap \{\{u,v\} | u, v \in V'\}$
21: **for** $\iota = i+1, i+2, \dots, j-1$ **do**
22: **if** $k_\iota \neq k_i$ **then**
23: $V' \leftarrow V' - \{u_\iota, v_\iota\}$; $E' \leftarrow E' \cap \{\{u,v\} | u, v \in V'\}$
24: **let** C be a shortest path between u_i and v_j in $G' = (V', E')$
25: **if** C is defined **and** $|C| < l$ **then return** C
26: **return** []

Notice also that the gradual generalization was adopted in the description of redundancies. Although long sequences subsume both less general redundancies, it

is not advisable to apply their replacement directly. It is better to apply elimination of redundancies stepwise from the less general one to more general ones. The reason for this practice is the increasing time complexity of redundancy elimination algorithms. A sequence of moves submitted to the more complex algorithm is potentially shortened by eliminating less general redundancies by following this practice.

5. Experimental Evaluation

An experimental evaluation was made with above three suggested methods for redundancy elimination.

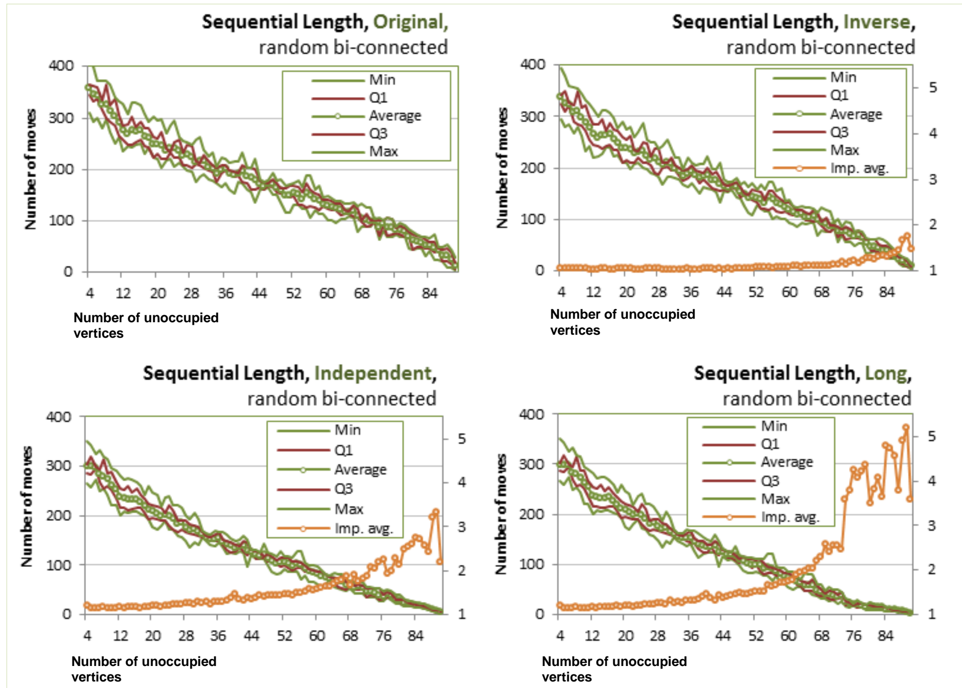


Fig. 2. *Sequential* ***length distribution*** *on random bi-connected graphs.* A collection of 10 graphs consisting of 90 vertices with length of handles ranging uniformly between 2 and 8 were generated for each number of unoccupied vertices. Minimum, maximum, average, first quartile, and third quartile out of sequential solution lengths of random instances over graphs from the collection are shown. The above characteristics of the solution length distribution are shown for original solutions as well as for solutions after removal of redundancies by the selected technique. The average improvement of solution is shown too in the same chart. It is possible to observe that solution lengths are distributed in a relatively narrow zone around the average length (approximately $\pm 10\%$ of the average length). The zone tends to narrow yet more for more sophisticated redundancy elimination.

Algorithms 1, 2, and 3 were implemented in C++ and were tested on a set of benchmark instances of PMG. Solutions found by the *BIBOX-θ* [11] algorithm on these benchmark instances were submitted to redundancy elimination methods.

Several characteristics of redundancy elimination were evaluated: the reduction of the total number of moves within solutions, parallel makespan, average parallelism, and runtime were measured. The implementation of redundancy elimination algorithms almost exactly follows the pseudo-code given in the previous section.

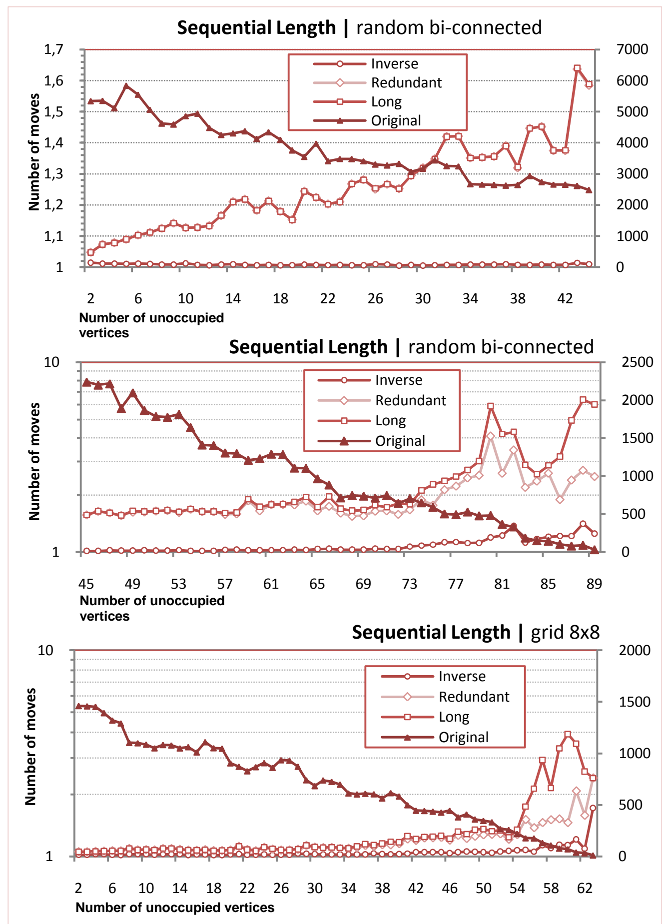


Fig. 3. ***Solution length*** *improvement on random bi-connected graph and 8×8 grid.* The total number of moves of the original solution and improvement ratio after applying redundancy elimination techniques are shown. As the number of unoccupied vertices grows the better improvements can be achieved. Up to 5 times smaller solutions can be obtained.

It was always the case that solution was processed by the less general redundancy elimination before it was submitted to more general and more sophisticated one. This measure ensures that the more time consuming algorithms obtains already processed solution for which there is a chance to be significantly shorter. The complete source code to allow reproducibility of all the experiments presented in this paper and raw experimental data are provided at the website: http://ktiml.mff.cuni.cz/~surynek/research/j-redundancy-2012.

Two structurally different sets of instances of the problem of PMG were tested. The first set of problems consists of randomly generated bi-connected graphs with approximately 90 vertices. The initial and the goal arrangement of pebbles were generated as a random permutation. The construction of the random bi-connected graphs exploits the construction that starts with a cycle followed by a gradual addition of handles to the currently constructed graph. Specifically, graphs were constructed by adding handles of random length (uniform distribution from interval 2..8) to the initial cycle of length 7. Tests were done with a collection of 10 different random bi-connected graphs of the above setup.

The second set of testing instances consists of a grid of the size 8×8 where the initial and the goal arrangement of pebbles were again random permutations.

The series of results presented in Fig. 2 are devoted to an evaluation of the distribution of the total number of moves within the solution on random bi-connected graphs. All the three redundancy elimination methods were evaluated in this test. The solution length is shown in the dependence on the number of unoccupied vertices which ranged from 4 to 89. The following characteristics calculated out of solution lengths for instances over the mentioned collection of 10 graphs are shown for each number of unoccupied vertices: maximum, minimum, first quartile, third quartile, and average length.

It can be observed from results in Fig. 2 that the sequential solution lengths tend to be close to the average solution length; more precisely they are in the zone of approximately $\pm 10\%$ around the average length from which it can be concluded that the original *BIBOX-θ* and redundancy elimination techniques have a stable behavior.

To keep the results readable the remaining results are presented for a single bi-connected graph only – one of those 10 randomly generated bi-connected graphs was chosen.

The reduction of the total number of moves within the solution depending on the increasing number of unoccupied vertices is shown in Fig. 3. It can be observed from Fig. 3 together with Fig. 2 that up to 5 times smaller solution can be obtained by applying redundancy elimination. The most expensive elimination of

long sequences is beneficial when there is approximately 70% and more unoccupied vertices.

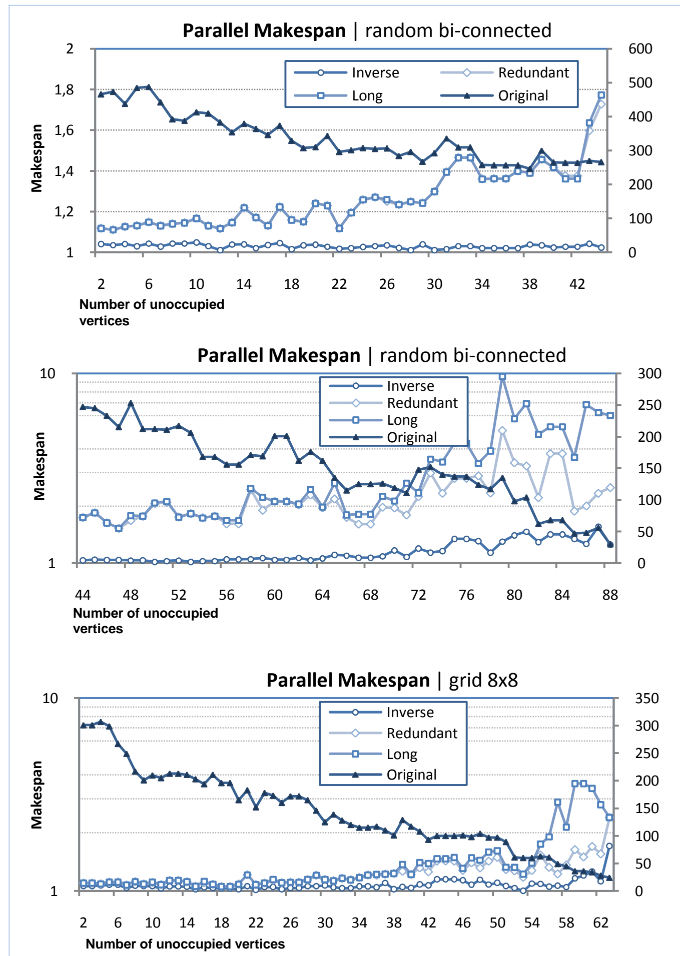


Fig. 4. *Parallel **makespan** improvement.* Redundancy elimination has even better effect on the makespan than on the size of the solution. Removal of redundancies allows more efficient increasing of the parallelism. Up to 10 times shorter solutions can be obtained on bi-connected graphs.

Results regarding the effect of redundancy elimination on parallel makespan are shown in Fig. 4. These results correlate well with the total number of moves while the improvement is slightly better for the makespan.

This observation is further quantified in Fig. 5. where the dependence of the average parallelism (which is defined as the total number of moves divided by the makespan) on the number of unoccupied vertices is shown. It can be observed that

redundancy elimination typically leads to a slight increase in the average parallelism.

Results regarding runtime on a testing machine are summarized in Fig. 6. Expectably, the runtime consumed to eliminate long sequences is highest while it is still reasonable for an offline post-processing. Eliminating inverse moves and redundant sequences is relatively cheap so they can be used as an on-line post-processing tool.

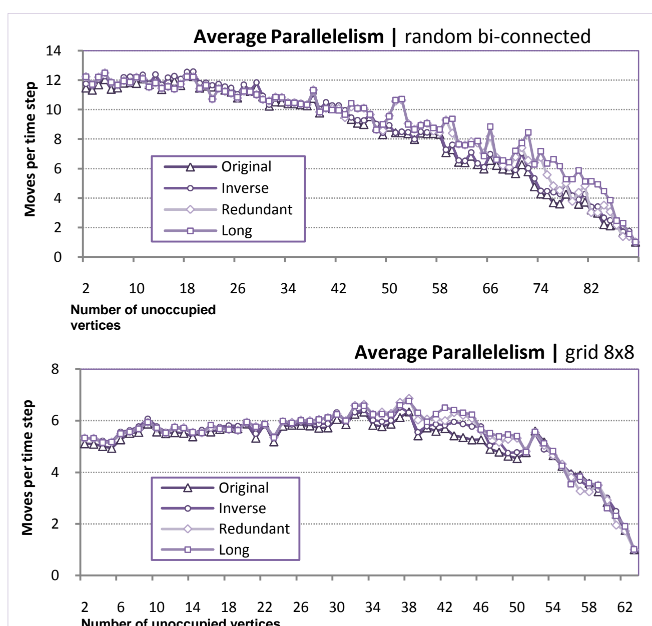


Fig. 5. *Average parallelism (average number of mover per time step).* The redundancy elimination leads to increasing of the parallelism most significantly when there is 50% to 90% of unoccupied vertices in the graph.

The last part of the results presented in Fig. 7 is devoted to an investigation of step parallelism – that is, the number of moves performed simultaneously at the individual time steps. A single random bi-connected graph used in previous tests is presented here as well. There were 60 vertices out of 90 unoccupied. Although it is difficult to make any analysis of such results, one aspect is quite apparent from presented results – it can be observed that the qualitatively most significant change occurs when the elimination of redundant moves is used (this observation has been done also on other graphs and setups which are not presented here). On the other hand, the change obtained by applying elimination of inverse moves on the original solution as well as the change obtained by eliminating long sequences

of moves from the solution which is already free of redundant moves is relatively little.

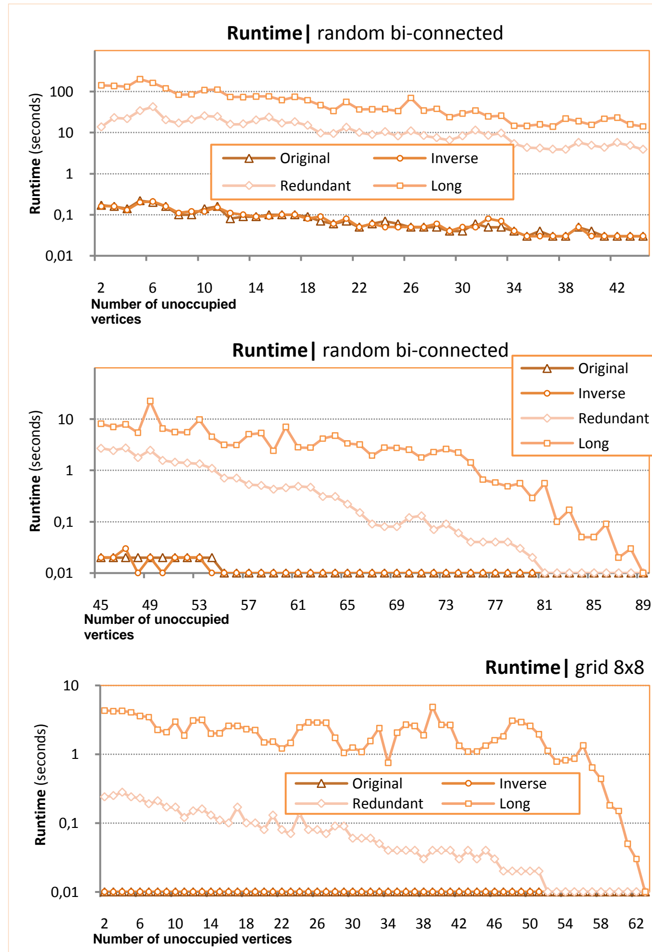


Fig. 6. *Runtime necessary for eliminating redundancies.* Eliminating long sequences is computationally the most costly (test were run on an Pentium 4, 2.4GHz, 512MB RAM, under Mandriva Linux 10.1, 32-bit edition).

It is possible to conclude that the solution can be improved by up to the order of magnitude in the measured characteristics for both types of tested graphs.

Removal of redundant sequences represents the best trade-off between detection cost and solution improvement according to performed experiments. Whereas eliminating inverse moves or long sequences feature extreme situations; the former brings almost no improvement; the latter seems to be computationally too costly for an on-line post-processing.

An expectable result is that the better improvement of solutions is gained when there are more unoccupied vertices in the input graph. Notice that definitions of redundancies are based on the mutual non-interfering of motions of pebbles. The more unoccupied space is available in the graph the less interference between moves of pebbles is possible.

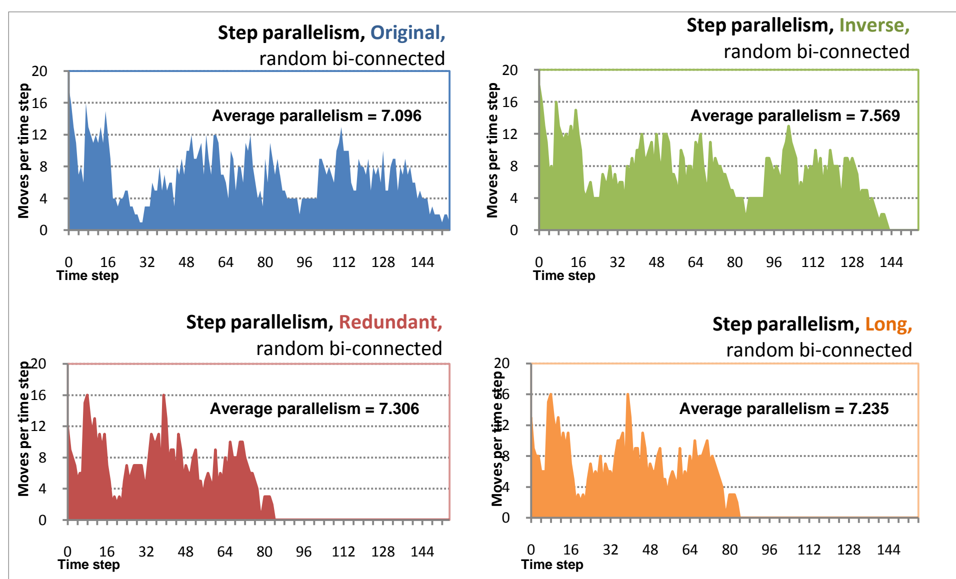


Fig. 7. *Step parallelism on random bi-connected graph.* The graph consists of 90 vertices and 60 of them are unoccupied. The length of handles was uniformly generated from the range 2..10 - the same setup as in other experiments. Number of moves in the individual time steps is shown.

6. SAT Based Solution Improvements: An Overview

Our novel solution optimization technique called COBOPT employs SAT solving technology [1] to optimize the solution with respect to the makespan. The technique has been suggested in [12]. To be able to use SAT solvers in this way we need to obtain some (sub-optimal) solution to the PMG instance first. Let this initial solution be called a *base solution*. In this regard we used the same original solution as the base solution as in the case of redundancy elimination methods.

The crucial building block for using SAT solving technology is an encoding of motion coordination instance as an instance of propositional satisfiability. That is, we need to build a propositional formula such that it is satisfiable if and only if a solution of a certain makespan to the given motion coordination instance exists. Suppose that we are given makespan ξ. We model the arrangements of pebbles at

every time step $1,2,\ldots,\xi$ where the arrangement at time step 1 is equal to the initial state and the arrangement at time step ξ is equal to the goal state. The individual arrangement consists of vectors of propositional variables for each vertex of G such that it tells us what pebble is located in the given vertex. Constraints to enforce valid transitions between consecutive time steps are also added. This encoding will be referred to as an *inverse encoding* in experiments.

Having such a propositional formula we are able to solve the given *solvable* PMG problem optimally with respect to the makespan. This is done by asking if a solution of some makespan ξ exists, where ξ is selected according to some search strategy. This asking strategy may be based for example on binary search – actually this is a strategy we use.

Notice that it is not possible to check that there is no solution to the PMG instance using this technique. However, as we use the technique to replace sub-optimal sub-solutions in the already constructed base solution we always know that the instance is solvable.

Algorithm 4. COBOPT: SAT-based PMG solution optimization – basic scheme based on binary search.

function *COBOPT-Optimize-Motion-Coordination-Plan* $(\mathrm{G},\vec{s},k^+)$: **solution**
1: $\vec{s}_+ \leftarrow \vec{s}$
2: **do**
3: $\quad \vec{s}_- \leftarrow \vec{s}_+$
4: $\quad$ **let** $\vec{s}_- = [S_P^1,\ldots,S_P^m]$
5: $\quad t \leftarrow 0;\ \vec{s}_+ \leftarrow []$
6: $\quad$ **while** $t < m$ **do**
7: $\quad\quad t^+ \leftarrow$ *Find-Last-Reachable-Arrangement*$(\mathrm{G},S_P^t,\vec{s}_-,k^+)$
8: $\quad\quad \vec{s}_+ \leftarrow \vec{s}_+$.*Compute-Optimal-Solution*$(\mathrm{G},S_P^t,S_P^{t^+})$
9: $\quad\quad t \leftarrow t^+$
10: **while** $|\vec{s}_-| > |\vec{s}_+|$
11: **return** $\vec{s}_+$

function *Find-Last-Reachable-Arrangement* $(\mathrm{G},S_P^t,\vec{s},k^+)$: **integer**
12: **let** $\vec{s} = [S_P^1,\ldots,S_P^m]$
13: $l \leftarrow t;\ u \leftarrow m+1$
14: **while** $u - l > 1$ **do**
15: $r \leftarrow (u+l)/2$
16: $\quad k \leftarrow \min(m - t, k^+)$
17: $\quad \Xi \leftarrow$ *Encode*(G,S_P^t,S_P^r,k)
18: $\quad$ **if** *Solve-SAT* (Ξ) **then** $l \leftarrow r$
19: $\quad$ **else** $u \leftarrow r$
20: **return** l

After producing a base solution, this is submitted to a SAT based optimization process. A maximum bound k^+ for encoding coordination instances is specified.

Then sub-sequences in the base solution are replaced with computed optimal sub-solution. Suppose that we are currently optimizing at time step t. It is computed what is the largest $t^+ > t$ such that the time step t^+ can be reached from the time step t with no more than k^+ steps. Then sub-solution of the base solution from the time step t to t^+ is replaced by the optimal one obtained from the SAT solver. The process then continues with optimization at time step t^+ until the whole base solution is processed.

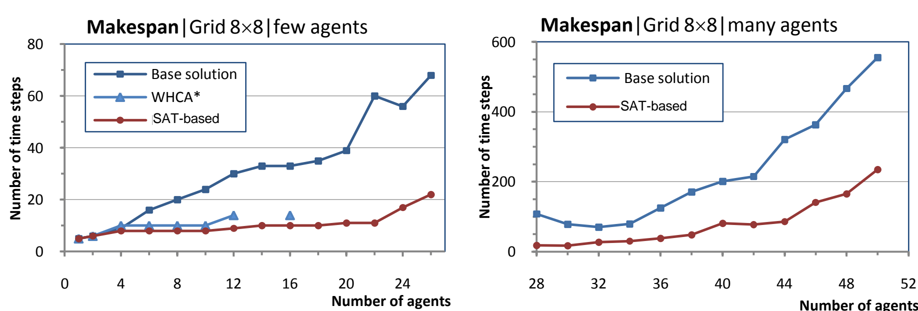


Fig. 8. *Makespan comparison on the 8×8 grid.* Optimal solutions for up to 22 and 30 agents can be found by SAT based optimization. Only up to 16 agents can be solved sub-optimally by WHCA*. The timeout for SAT based optimization was 3600 seconds.

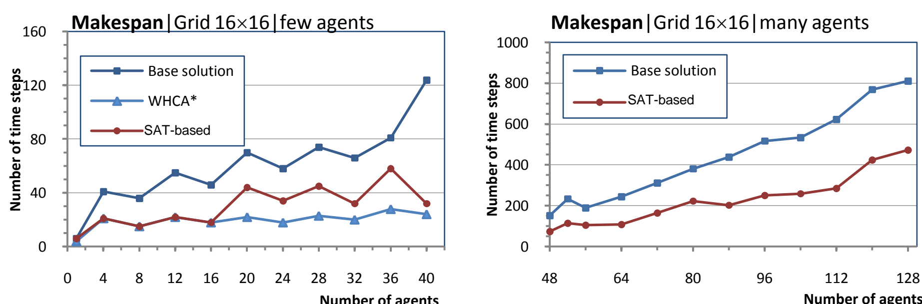


Fig. 9. *Makespan comparison on the 16×16 grid.* Optimal solutions for up to 40 agents can be found by SAT-based optimization; in the same range WHCA* can find near optimal solution as well. The timeout for SAT based optimization was 3600 seconds.

The optimization process can be iterated by taking new solution as the base one until a fixed point is reached. The binary search is employed to find t^+ and the optimal sub-solution in order to reduce the number of SAT solver invocations – see Algorithm 4. which summarizes basic COBOPT optimization method formally.

Notice that separation points in the base solution are selected on the greedy basis – optimization always continues on the first not yet processed time step. We

also considered optimizing placement of separation point by dynamic programming techniques. This approach generates slightly better base solution decomposition. However it is at the great expense in overall runtime as many more invocation of the SAT solver are necessary.

In the experimental evaluation with SAT based optimization of solutions we also made comparison with the WHCA* algorithm [9] that is known to generate solutions that have makespan near to the optimum. WHCA* is however not able to tackle instances with environments densely occupied by agents.

Results showing comparison of the SAT-based optimization with respect to the base solution as well as with respect to WHCA* on 4-connected grids are shown in Fig. 8 and Fig. 9. The time limit for optimization was set to 3600 seconds. The process either found an optimal solution or the time limit was reached. It can be observed that SAT based optimization generates better solutions than WHCA*. Optimal solutions were obtained in cases with few agents.

If we compare SAT-based optimization with redundancy elimination methods it can be stated that SAT-based optimization is more general. It is able to discover a redundancy of a priori unknown type. On the other SAT based optimization is more time consuming which makes it suitable for off-line solving of the problem only while redundancy eliminations can be used on-line. Lot of improvements in the makespan when SAT based optimization is used comes from increasing parallelism – more moves are performed per single time step. It may happen that even though makespan of the solution has been improved the number of moves within the solution may increase.

7. Summary, Conclusions, and Future Work

This work addressed the quality (makespan) of solutions of problems motion coordination. Particularly, solutions generated by the existing algorithm *BIBOX-θ* for the given class of the problem were analyzed with respect to the presence of certain type of redundancies. Our hypothesis was that there exist certain types redundancies in generated solutions while we were not aware how do they look like.

A special visualization tool *GraphRec* was used for analyzing solutions produced by the *BIBOX-θ* algorithm. This tool allowed automating two tasks that cannot be made manually – proper drawing of a graph which a given instance consists of and visualizing moves of entities over this graph. The tool eventually confirmed that redundancies really exist and it was possible to propose their formal description.

Several types of redundancies were defined and methods for their elimination were proposed. To justify quality of our proposal an extensive experimental evaluation of proposed methods was performed on the number of different problem setups. It eventually confirmed that solutions can be improved by up to the order of magnitude using the suggested methods. The secondary finding is that the better improvement can be gained for problems with higher number of unoccupied vertices.

As a next step in solution improvements we suggest to employ SAT solving technology. A propositional formula satisfiable if and only if a given instance of motion coordination problem is solvable within the given makespan is constructed. Such a formula allows asking what is the makespan optimal replacement for a given sub-solution of an existing solution. The solution improvement process then repeatedly replaces sub-solutions by optimal ones until time limit is reached or the makespan optimal solution is found.

The SAT based technique generates high quality solutions with respect to the makespan however it is very time consuming. Thus it is more suitable for off-line improvements of solutions. On the other hand redundancy elimination methods are fast enough and can be used on-line.

References

1. N. **Eén**, N. **Sörensson**, "An Extensible SAT-solver," Proceedings of Theory and Applications of Satisfiability Testing (SAT 2003), pp. 502-518, LNCS 2919, Springer, 2004.

2. T. H. **Cormen**, C. E. **Leiserson**, R. L. **Rivest**, C. **Stein**, "Introduction to Algorithms (Second edition)," MIT Press and McGraw-Hill, 2001, ISBN 0-262-03293-7.

3. A. **Kishimoto**, N. R. **Sturtevant**, "Optimized algorithms for multi-agent routing," Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Volume 3, IFAAMAS 2008, pp. 1585-1588.

4. D. **Kornhauser**, G. L. **Miller**, P. G. **Spirakis**, "Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications," Proceedings of the 25th Annual Symposium on Foundations of Computer Science (FOCS 1984), IEEE Press, 1984, pp. 241-250.

5. P. **Koupý**, "GraphRec - a visualization tool for entity movement on graph," Student project web page, http://www.koupy.net/graphrec.php, 2011, (January 2011).

6. I. **Parberry**, "A real-time algorithm for the (n^2-1)-puzzle," Information Processing Letters, Volume 56 (1), pp. 23-28, Elsevier, 1995.
7. D. **Ratner** and M. K. **Warmuth**, "Finding a Shortest Solution for the N×N Extension of the 15-PUZZLE Is Intractable," Proceedings of the 5th National Conference on Artificial Intelligence (AAAI 1986), Morgan Kaufmann Publishers, 1986, pp. 168-172.
8. M. R. K. **Ryan**, "Exploiting subgraph structure in multi-robot path planning," Journal of Artificial Intelligence Research (JAIR), Volume 31, (January 2008), AAAI Press, 2008, pp. 497-542.
9. D. **Silver**, "Cooperative Pathfinding," Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2005), AAAI Press, 2005, pp. 117-122.
10. P. **Surynek**, "A Novel Approach to Path Planning for Multiple Robots in Biconnected Graphs," Proceedings of the 2009 IEEE International Conference on Robotics and Automation (ICRA 2009), IEEE Press, 2009, pp. 3613-3619.
11. P. **Surynek**, "An Application of Pebble Motion on Graphs to Abstract Multirobot Path Planning," Proceedings of the 21st International Conference on Tools with Artificial Intelligence (ICTAI 2009), IEEE Press, 2009, pp. 151-158.
12. P. **Surynek**, "Towards Optimal Cooperative Path Planning in Hard Setups through Satisfiability Solving", Proceedings of PRICAI 2012, LNCS 7458, Springer, 2012, pp. 564-576.
13. R. E. **Tarjan**, "Depth-First Search and Linear Graph Algorithms," SIAM Journal on Computing, Volume 1 (2), pp. 146-160, Society for Industrial and Applied Mathematics, 1972.
14. K. C. **Wang** and A. **Botea**, "Tractable Multi-Agent Path Planning on Grid Maps," Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009), IJCAI Conference, 2009, pp. 1870-1875.
15. J. **Westbrook**, R. E. **Tarjan**, "Maintaining bridge-connected and bi-connected components on-line," Algorithmica, Volume 7, Number 5&6, pp. 433–464, Springer, 1992.
16. R. M. **Wilson**, "Graph Puzzles, Homotopy, and the Alternating Group," Journal of Combinatorial Theory, Ser. B 16, Elsevier, 1974, pp. 86-96.

Pavel **Surynek**: *Pre-processing in Boolean Satisfiability Using Bounded (2,k)-Consistency on Regions with Locally Difficult Constraint Setup.* International Journal on Artificial Intelligence Tools (IJAIT), Volume 23, Number 01 (29 pages), World Scientific, 2014, ISSN 0218-2130.

Preprocessing in Propositional Satisfiability Using Bounded *(2,k)*-Consistency on Regions with a Locally Difficult Constraint Setup

Pavel Surynek

Charles University Prague
Faculty of Mathematics and Physics
Department of Theoretical Computer Science and Mathematical Logic
Malostranské náměstí 25, Praha, 118 00, Czech Republic

Graduate School of Maritime Sciences, Intelligent Informatics Laboratory
Kobe University, Japan
5-1-1 Fukae-minamimachi, Higashinada-ku, Kobe 658-0022, Japan

pavel.surynek@mff.cuni.cz

Abstract. A new type of partially global consistency derived from $(2,k)$-consistency called bounded $(2,k)$-consistency (B2C-consistency) is presented in this paper. It is designed for application in propositional satisfiability (SAT) as a building block for a preprocessing tool. Together with the new B2C-consistency a special mechanism for selecting regions of the input SAT instance with difficult constraint setup was also proposed. This mechanism is used to select suitable difficult sub-problems whose simplification through consistency can lead to a significant reduction in the effort needed to solve the instance. A new prototype preprocessing tool `preprocessSIGMA` which is based on the proposed techniques was implemented. As a proof of new concepts a competitive experimental evaluation on a set of relatively difficult SAT instances was conducted. It showed that our prototype preprocessor is competitive with respect to the existent preprocessing tools `LiVer`, `NiVer`, `HyPre`, blocked clause elimination (`BCE`), and `Shatter` with `saucy 3.0`.

***Keywords*:** SAT; CSP; SAT preprocessing; local consistency; global consistency; $(2,k)$-consistency; probability; difficult instances; hyper-resolution; blocked clause elimination; symmetry

1. Introduction and Motivation

Recent works dealing with difficult instances of *propositional satisfiability* (*SAT*) [1, 2, 8, 10, 26] indicate that an intelligent preprocessing focused on the structure of an instance can dramatically reduce the effort needed to solve it. Technically, the preprocessing task is done by transforming the input instance into another one (hopefully simpler), which is subsequently submitted to a general purpose *SAT solver* [8, 13]. It is crucial that the preprocessing step is fast enough relative to the runtime of the SAT solver on the preprocessed instance.

In this work, we further develop ideas from [26] where the input propositional formula is interpreted as a graph, in which graph structures – namely complete sub-graphs – are identified and, after some calculation involving the number and the size of complete sub-graphs, an inference is made. The drawback of the original idea from [26] is that it requires the input instance to be relatively well structured to be able to identify acceptable complete sub-graph decomposition. In this paper, we overcome this major drawback using two new techniques. First, a **new type of consistency** derived from $(2,k)$*-consistency* [11] called *bounded* $(2,k)$*-consistency with complete graphs* (*B2C-consistency*) is proposed. It uses graph interpretation of a sub-problem on which reasoning over its decomposition into complete sub-graphs is performed and can therefore be regarded as a partially global reasoning mechanism. Second, a **new mechanism for selecting** a sub-problem suitable for applying the consistency is proposed. In order to maximize the benefit of inferences made through consistency, we proposed to apply it on regions of the input instance with a locally difficult constraint setup. It means that we are trying to choose such a sub-problem for applying the consistency that, in itself, is difficult in a certain sense (focusing on the difficulty proved to be beneficial in [26] but the previous technique required the whole instance to exhibit a difficult constraint setup). We were primarily inspired by the difficulty of well known problems such as the *pigeon/hole principle* (*P/H principle*) or *FPGA routing* [1, 2] and we are trying to select regions of the instance which, in terms of certain properties, are similar to these difficult instances. To do this, a characteristic called the *expected number of satisfied tuples of values* is used so that regions that have this characteristic similar to difficult instances are used as sub-problems on which B2C-consistency is applied. In this way, we are able to discover sub-problems with a hidden difficulty and simplify them with the proposed consistency reasoning, which provides a faster solution of the output instance.

As a validation of the proposed concepts a prototype SAT preprocessing tool `preprocessSIGMA` [27] based on B2C-consistency and a new sub-problem selection technique have been implemented. The performed experimental evalua-

tion showed that our prototype preprocessing tool is competitive with respect to existent prominent preprocessing tools such as `LiVer` [25], `NiVer` [25], `HyPre` [5], *blocked clause elimination* [16, 20] (`precosat-465`), and `Shatter` with `saucy 3.0` [2, 21] which is so far the latest version.

This work has been iteratively developed and preceding work related to the presented one appeared in [26]. The organization of the paper is as follows: basic concepts from *constraint programming* [11] and SAT are introduced in **Section 2**. The concept of B2C-consistency is subsequently developed (**Section 3**). The following section (**Section 4**) deals with the question of how to build a preprocessing tool exploiting B2C-consistency. Finally (**Section 5**), an extensive experimental evaluation focused on the competitiveness and the investigation of internal properties of the implemented preprocessor is presented.

2. Background from Constraint Programming and Propositional Satisfiability

Let us start with the basic notation and definitions used in the rest of the paper. This section represents the basic background from *constraint programming* [11] and *propositional satisfiability* [8], which the new concepts rely on.

Tuples and *lists* (that is, sequences) consisting of some objects will be denoted using brackets (for example $[x, y]$ denotes an ordered pair consisting of two objects x and y; [] denotes the empty list).

Definition 1 ***(Constraint Satisfaction Problem)*** [11]**.** A *constraint satisfaction problem* (*CSP*) over a given finite universe $\mathbb{D}$ is a triple (X, D, C) where X is a finite set of *variables*, C is a finite set of *constraints*, and $D: X \longrightarrow 2^{\mathbb{D}}$ is a function assigning each variable a finite *domain*. A constraint $c \in C$ is a construct of the form $\langle [x_1^c, x_2^c, \dots, x_{a^c}^c], R^c \rangle$ where $a^c \in \mathbb{N}$ is the *arity* of constraint c, $[x_1^c, x_2^c, \dots, x_{a^c}^c]$ with $x_i^c \in X$ for $i = 1,2, \dots, a^c$ is called a *scope* of c, and $R^c \subseteq D(x_1^c) \times D(x_2^c) \times \dots \times D(x_{a^c}^c)$ is a relation that enumerates a set of tuples of values for which constraint c is satisfied. □

For simplicity, it is sometimes assumed that $D(x) = \mathbb{D}$ for every $x \in X$. We will use this assumption as well in certain cases. Furthermore, it is assumed that we can reorder variables in the scope of a constraint arbitrarily using the above notation. For example, if there is a constraint $c = \langle [x, y], R^c \rangle$ in C, we can suppose that there is also an equivalent formulation of c as a constraint $e = \langle [y, x], R^e \rangle$ in C where relation R^e can be obtained from R^c by swapping its components.

Definition 2 ***(Solution of CSP)*** [11]. An assignment $v: X \longrightarrow \mathbb{D}$ such that $v(x) \in D(x)$ for every $x \in X$ is called a solution of a given CSP (X, D, C) if it is defined for every variable in X and all the constraints in C are satisfied by v. That is, it holds that $[v(x_1^c), v(x_2^c), \dots, v(x_{a^c}^c)] \in R^c$ for every constraint $c = \langle [x_1^c, x_2^c, \dots, x_{a^c}^c], R^c \rangle \in C$. □

Regarding constraints, we will sometimes use a formulation that some tuple of values is allowed/forbidden by a constraint, which means exactly that the tuple belongs or does not belong to the defining relation of the constraint.

Closely related to CSP is the *propositional satisfiability problem* (*SAT*) [8, 10]. It is introduced in the following two definitions. Note that in CSP we are trying to find a valuation of variables such that **all** the constraints are satisfied (that is, the conjunction of all the constraints is satisfied). In SAT the task is similar. We are trying to find a propositional valuation that satisfies **all** the clauses of the input formula (the formula has typically the form of a conjunction of clauses – CNF).

Definition 3 ***(Propositional Formula)*** [10]. A *propositional formula* in the *conjunctive normal form* (*CNF*) over a given set of propositional variables Ω is a conjunction: $\bigwedge_{i=1}^{n} \Gamma_i$ where $n \in \mathbb{N}_0$ and each Γ_i with $i \in \{1,2,\dots,n\}$ is a *clause* that puts into a disjunction *literals* over variables from Ω. That is, $\Gamma_i = \bigvee_{k=1}^{\alpha^i} \psi_k^i$ for $i = 1,2,\dots,n$ where $\alpha^i \in \mathbb{N}$ is the size of the clause and either $\psi_k^i = \beta$ or $\psi_k^i = \neg\beta$ for some variable $\beta \in \Omega$ for every $k = 1,2,\dots,\alpha^i$. □

Definition 4 ***(Propositional Satisfiability Problem)*** [10]. A *valuation* of propositional variables is an assignment $\omega: \Omega \longrightarrow \{FALSE, TRUE\}$. The given valuation of variables ω can be naturally extended to a valuation of formulae over Ω denoted as ω^*. A *propositional satisfiability problem* (*SAT*) with a formula Φ over Ω is the task of determining whether there exists a valuation ω of Ω such that $\omega^*(\Phi) = TRUE$. □

We are about to work with the concept of *consistencies* [11] in SAT which is, however, the concept from constraint programming used over CSPs. Hence, it is convenient to define translation of SAT to CSP so that we are able to work with consistencies in SAT through this translation. For this purpose, we chose the so-called *literal encoding* [6, 27] which provides such a translation in the natural way.

Definition 5 ***(Literal Encoding of SAT)*** [27]. Let $\Phi = \bigwedge_{i=1}^{n} \Gamma_i$ with $\bigvee_{l=1}^{\alpha^i} \psi_l^i$ for $i = 1,2,\dots,n$ be a propositional formula in CNF over Ω. A *literal encoding* of Φ is a CSP $E^0(\Phi) = (X_\Phi^0, D_\Phi^0, C_\Phi^0)$ where $X_\Phi^0 = \{\bar{\Gamma}_1, \bar{\Gamma}_2, \dots, \bar{\Gamma}_n\}$, $D_\Phi^0(\bar{\Gamma}_i) = \{\bar{\psi}_l^i | l = 1,2,\dots,\alpha^i\}$ for every $i = 1,2,\dots,n$; and there are constraints between all the pairs of variables as follows: $[\bar{\psi}_l^i, \bar{\psi}_t^j]$ where $\bar{\psi}_l^i \in D_\Phi^0(\bar{\Gamma}_i)$ and $\bar{\psi}_t^j \in D_\Phi^0(\bar{\Gamma}_j)$ is forbidden by relation R^c defining constraint $c = \langle [\bar{\Gamma}_i, \bar{\Gamma}_j], R^c \rangle$ with $i,j \in \{1,2,\dots,n\}$, $l \in \{1,2,\dots,\alpha^i\}$, and $t \in \{1,2,\dots,\alpha^j\}$ if there is $\beta \in \Omega$ such that either $\beta = \psi_l^i$ and $\neg\beta = \psi_t^j$ or $\neg\beta = \psi_l^i$ and $\beta = \psi_t^j$. □

The stripe above the generic symbols is used to distinguish constant symbols (with the stripe) which do not evaluate from variables (without the stripe) which do evaluate (down to other constants). Note that literal encoding is a *binary CSP*; that is, all the constraints have arity of at most 2.

For our purposes, literal encoding is further processed to capture constraints imposed by the original formula more explicitly (note that there is an incompatibility between complementary literals only at this stage). A new incompatibility is introduced as a constraint between every two literals ψ_l^i and ψ_t^j with $i,j \in \{1,2,\dots,n\}$ such that $i \neq j$, $l \in \{1,2,\dots,\alpha^i\}$ and $t \in \{1,2,\dots,\alpha^j\}$ if the *singleton unit propagation* [12, 26] with the setting $\psi_l^i = TRUE$ infers that $\psi_t^j = FALSE$ with respect to Φ (that is, it is set that $\psi_l^i = TRUE$; all the other variables are left unassigned and unit propagation follows). Let this modification of literal encoding be called an *explicit literal encoding* and it will be denoted as $E^1(\Phi) = (X_\Phi^1, D_\Phi^1, C_\Phi^1)$ (the upper index implies that the first stage of inference has been made).

We are now ready to define the so-called $(2,k)$-consistency [11]. It is a generalization of k-consistency [24] which checks whether a value is supported by a k-tuple of values from the domains of other variables. Within $(2,k)$-consistency, it is checked whether a pair of consistent values has a supporting k-tuple of values. If there is no such supporting k-tuple of values the value or the pair of values respectively can be ruled out from further consideration by an additional constraint.

An auxiliary operation of *projection* denoted as $T|_{A \to B}$ will be used to transform a tuple T into another tuple with respect to patterns A and B. Tuple T and pattern A are of the same size and B is contained by A. The result of the projection is obtained by matching pattern A on T followed by selecting components of T associated with their counterparts in A that correspond to B (for stance, $[1,2,3]|_{a,b,c \to c,b} = [3,2]$).

Definition 6 *($(\mathbf{2},\boldsymbol{k})$-Consistency)* [11]. Let $k \in \mathbb{N}$ be a natural number, (X, D, C) be a CSP, and $x_0, x_1, \ldots, x_k, x_{k+1} \in X$ be a $(k+2)$-tuple of distinct variables. A pair of values $d_0 \in D(x_0)$ and $d_{k+1} \in D(x_{k+1})$ with $[d_0, d_{k+1}] \in R^c$ for every binary constraint $c = \langle [x_0, x_{k+1}], R^c \rangle$ in C is called to be $(2, k)$-*consistent* with respect to k-list of variables $x_1, x_2, \ldots, x_k$ if there exists a k-tuple of values $d_1 \in D(x_1)$, $d_2 \in D(x_2)$,..., $d_k \in D(x_k)$ such that for every constraint $e = \langle [z_1^e, z_2^e, \ldots, z_{a^e}^e], R^e \rangle$ in C with $\{z_1^e, z_2^e, \ldots, z_{a^e}^e\} \subseteq \{x_0, x_1, \ldots, x_{k+1}\}$ it holds that $[d_0, d_1, \ldots, d_{k+1}]|_{x_0, x_1, \ldots, x_{k+1} \to z_1^e, z_2^e, \ldots, z_{a^e}^e} \in R^e$. The pair of values $d_0 \in D(x_0)$ and $d_{k+1} \in D(x_{k+1})$ is called to be $(2, k)$-*consistent* if it is $(2, k)$-consistent with respect to all the k-tuples of variables $x_1, \ldots, x_k \in X$. Finally, CSP (X, D, C) is called to be $(2, k)$-*consistent* if all the pairs of values from domains of every two distinct variables are $(2, k)$-consistent. □

It is not difficult to see that checking whether there exists a supporting k-tuple of values with respect to a fixed k-list of variables of unbounded size k is an *NP-complete problem* [22] in both k-consistency and $(2, k)$-consistency (for example, the *graph coloring problem* can be reduced to the task of searching for a supporting k-tuple). Hence, unless $P = NP$, the support cannot be found in polynomial time.

Another simple observation is that a support with respect to a fixed list of variables can be found in $O(|\mathbb{D}|^k)$ by traversing all the involved k-tuples of values. This is also the currently best known upper bound of the time complexity of the search for a support within $(2, k)$-consistency enforcing algorithms [11].

Both the discussed higher level consistencies represent powerful techniques when k is bounded by the number of variables only. After enforcing k-consistency/$(2, k)$-consistency with k high enough it is possible to obtain a solution of a problem in a backtrack-free manner [11]. Without providing more details, the high enough k means that it is at least the *width of the constraint graph* of the given CSP which does not exceed the number of variables [14].

3. Bounded $(2, k)$-Consistency with Complete Graphs – B2C Consistency

Our new concept of the so-called *bounded $(2, k)$-consistency with complete graphs* (*B2C-consistency*) combines the inference strength of $(2, k)$-consistency with graph-based global reasoning. The global oriented reasoning in SAT which is of our interest was first introduced in [26]. Particularly, the idea of exploiting global information reflected in complete sub-graphs in a certain graph interpretation of the problem has been taken from the previous work and further elaborated.

However, global reasoning itself turned out to be unilateral and hence not ideally suitable for using in SAT preprocessing. Therefore, it is suggested in this work to enhance global reasoning with $(2,k)$-consistency, which is quite universal and is supposed to help in cases where global reasoning alone is unsuitable. If both the approaches – global and $(2,k)$-consistency – are applied together a synergic effect is produced in certain situations.

Local consistencies such as k-consistency and related consistencies in SAT have been studied in several works [7, 23, 29]. The common approach in these works is to encode a given task so that a local consistency of interest is simulated by *unit propagation* [12]. Our approach takes an instance of SAT problem as a list of clauses (constraints) and applies the consistency directly without caring about the way how the original task has been encoded into the instance. The result is a set of forbidden value assignments in the case of *B2C-consistency* which is subsequently submitted to a SAT solver together with the original instance as a list of additional clauses.

The major obstacle with $(2,k)$-consistency is that it is difficult to enforce because it is necessary to search for a consistent k-tuple of values, which means to traverse the search space of the size of $|\mathbb{D}|^k$ in the worst case (supposed that all the variables have an identical domain of $\mathbb{D}$). Hence, to preserve low computation costs of the consistency enforcing algorithm we suggest to bound the consistency in some way. It has been chosen to bound the number of steps of the search for a consistent k-tuple by constant Λ.

B2C-consistency is again defined with respect to a $(k+2)$-tuple of distinct variables. Again, it checks whether a given pair of values from domains of two distinct variables have a supporting k-tuple in domains of remaining k variables. The following sections describe how the new consistency is enforced supposed that $(k+2)$-list of variables has been already determined. The process of selecting a promising $(k+2)$-tuple is discussed later.

3.1. *A Graph Derived from SAT – Graph Interpretation*

Let $E^1(\Phi) = (X_\Phi^1, D_\Phi^1, C_\Phi^1)$ be an explicit literal encoding of a given propositional formula Φ. Next, let us have $k \in \mathbb{N}$ and an ordered $(k+2)$-tuple of selected variables $K_+^2 = [\bar{\Gamma}_{i_0}, \bar{\Gamma}_{i_1}, \bar{\Gamma}_{i_2}, \dots, \bar{\Gamma}_{i_k}, \bar{\Gamma}_{i_{k+1}}] \subseteq X_\Phi^1$ with $i_0, i_1, \dots, i_{k+1} \in \{1,2,\dots,n\}$ where $i_\zeta \neq i_\xi$ for $\zeta, \xi \in \{0,1,\dots,k+1\}$ with $\zeta \neq \xi$.

It is more convenient to define consistency with respect to an undirected graph derived from the constraint network. A target undirected graph will be represented by the so-called *graph interpretation* in the given context. It is defined with respect to K_+^2 as an undirected graph $I(K_+^2) = (I_V, I_E)$ where a set of vertices I_V

consists of $\bigcup_{\zeta=0}^{k+1}\{\bar{\psi}_l^{i_\zeta} \,|\, l = 1,2,\dots,\alpha^{i_\zeta}\}$ and a set of edges I_E contains edge $\{\bar{\psi}_l^{i_\zeta}, \bar{\psi}_t^{i_\xi}\}$ with $\zeta, \xi \in \{0,1,\dots,k+1\}$ such that $\zeta \neq \xi$, $l \in \{1,2,\dots,\alpha^{i_\zeta}\}$, and $t \in \{1,2,\dots,\alpha^{i_\xi}\}$ if it holds that $[\bar{\psi}_l^{i_\zeta}, \bar{\psi}_t^{i_\xi}] \notin R^c$ for some constraint $c = \langle[\bar{\Gamma}_{i_\zeta}, \bar{\Gamma}_{i_\xi}], R^c\rangle$ in C_Φ^1 (edges stand for forbidden pairs of values; that is, an edge represents a *conflict*).

Propositional Formula Φ

$\Omega = \{\beta_1, \beta_2, \beta_3\}$

$\Phi =$

$\Gamma_1: (\ \beta_1 \vee \ \beta_2) \quad \wedge$

$\Gamma_2: (\neg\beta_1 \vee \neg\beta_2) \quad \wedge$

$\Gamma_3: (\ \beta_1 \vee \neg\beta_3) \quad \wedge$

Explicit Literal Encoding $E^1(\Phi)$

$E^1(\Phi) = (X_\Phi^1, D_\Phi^1, C_\Phi^1)\, X_\Phi^1 = \{\bar{\Gamma}_1, \bar{\Gamma}_2, \bar{\Gamma}_3, \bar{\Gamma}_4\}$

$D_\Phi^1(\bar{\Gamma}_1) = \{\bar{\beta}_1, \bar{\beta}_2\} \qquad D_\Phi^1(\bar{\Gamma}_2) = \{\neg\bar{\beta}_1, \neg\bar{\beta}_2\}$

$D_\Phi^1(\bar{\Gamma}_3) = \{\bar{\beta}_1, \neg\bar{\beta}_3\} \qquad D_\Phi^1(\bar{\Gamma}_4) = \{\neg\bar{\beta}_1, \bar{\beta}_2, \bar{\beta}_3\}$

$C_\Phi^1 = \{c_{\{1,2\}}, c_{\{1,3\}}, c_{\{1,4\}}, c_{\{2,3\}}, c_{\{2,4\}}, c_{\{3,4\}}\}$

$c_{\{1,2\}} = \langle[\bar{\Gamma}_1, \bar{\Gamma}_2], D_\Phi^1(\bar{\Gamma}_1) \times D_\Phi^1(\bar{\Gamma}_2) \setminus \{[\bar{\beta}_1, \neg\bar{\beta}_1], [\bar{\beta}_2, \neg\bar{\beta}_2]\}\rangle$

$c_{\{1,3\}} = \langle[\bar{\Gamma}_1, \bar{\Gamma}_3], D_\Phi^1(\bar{\Gamma}_1) \times D_\Phi^1(\bar{\Gamma}_3) \setminus \{[\bar{\beta}_1, \neg\bar{\beta}_3], [\bar{\beta}_2, \bar{\beta}_1]\}\rangle$

$c_{\{1,4\}} = \langle[\bar{\Gamma}_1, \bar{\Gamma}_4], D_\Phi^1(\bar{\Gamma}_1) \times D_\Phi^1(\bar{\Gamma}_4) \setminus \{[\bar{\beta}_1, \neg\bar{\beta}_1], [\bar{\beta}_1, \bar{\beta}_2], [\bar{\beta}_2, \bar{\beta}_3]\}\rangle$

$c_{\{2,3\}} = \langle[\bar{\Gamma}_2, \bar{\Gamma}_3], D_\Phi^1(\bar{\Gamma}_2) \times D_\Phi^1(\bar{\Gamma}_3) \setminus \{[\neg\bar{\beta}_1, \bar{\beta}_1], [\neg\bar{\beta}_2, \neg\bar{\beta}_3]\}\rangle$

$c_{\{2,4\}} = \langle[\bar{\Gamma}_2, \bar{\Gamma}_4], D_\Phi^1(\bar{\Gamma}_2) \times D_\Phi^1(\bar{\Gamma}_4) \setminus \{[\neg\bar{\beta}_1, \bar{\beta}_3], [\neg\bar{\beta}_2, \neg\bar{\beta}_1], [\neg\bar{\beta}_2, \bar{\beta}_2]\}\rangle$

Graph Interpretation $I(K_+^2)$

$K_+^2 = [\bar{\Gamma}_1, \bar{\Gamma}_2, \bar{\Gamma}_4] \qquad I(K_+^2) = (I_V, I_E)$

$I_V = \{\bar{\psi}_1^1, \bar{\psi}_2^1, \bar{\psi}_1^2, \bar{\psi}_2^2, \bar{\psi}_1^3, \bar{\psi}_2^3, \bar{\psi}_3^3\}$

$I_E =$

$\{\{\bar{\psi}_1^1, \bar{\psi}_1^2\}, \{\bar{\psi}_1^1, \bar{\psi}_1^4\}, \{\bar{\psi}_1^1, \bar{\psi}_2^4\},$

$\{\bar{\psi}_2^1, \bar{\psi}_2^2\}, \{\bar{\psi}_2^1, \bar{\psi}_3^4\},$

$\{\bar{\psi}_1^2, \bar{\psi}_3^4\},$

$\bar{\Gamma}_1$ $\bar{\Gamma}_2$ $\bar{\Gamma}_4$

$\bar{\psi}_1^1$ $\bar{\beta}_1$ $\bar{\psi}_2^1$ $\bar{\beta}_2$ $\bar{\psi}_1^2$ $\neg\bar{\beta}_1$ $\bar{\psi}_2^2$ $\neg\bar{\beta}_2$ $\bar{\psi}_1^4$ $\neg\bar{\beta}_1$ $\bar{\psi}_2^4$ $\bar{\beta}_2$ $\bar{\psi}_3^4$ $\bar{\beta}_3$

Figure 1. *Graph interpretation.* An original input Propositional formula Φ with four clauses is shown (upper left). Then a corresponding explicit literal encoding (upper right – that is, a literal encoding after singleton unit propagation) – the CSP model consisting of four variables is provided. The lower part depicts a graph interpretation over three variables selected in the CSP model. Dotted edges represent binary clauses that come from singleton unit propagation.

3.2. *Initial Setup of B2C-Consistency*

We are about to utilize structural information contained in the graph interpretation. It has been shown in the previous work [26] that useful structural information is constituted by the knowledge of complete constraint sub-graphs. Regarding

the given context, we can observe that **at most one literal** can be satisfied in a complete sub-graph in the graph interpretation of a literal encoding of a SAT instance. If a large enough complete sub-graph is detected in the graph interpretation, its knowledge can be used for an efficient search space pruning or a strong global inference. The exact process of doing so will be explained in detail in the following text.

A decomposition into complete sub-graphs of a given graph interpretation $I(K_{+}^{2}) = (I_V, I_E)$ is constructed first. It is a task of finding number $\delta \in \mathbb{N}$ and sets $I_V^1, I_V^2, \ldots, I_V^\delta \subseteq I_V$ called decomposition sets that satisfy the following conditions:

- (i) $\bigcup_{i=1}^{\delta} I_V^i = I_V$; that is, all the vertices are covered by the decomposition;
- (ii) $I_V^i \nsubseteq I_V^j$ for any two $i, j \in \{1,2,\ldots,\delta\}$ such that $i \neq j$; that is, the decomposition is not allowed to contain redundancies;
- (iii) I_V^i induces a complete sub-graph over edges from I_E for every $i \in \{1,2,\ldots,\delta\}$;
- (iv) $\forall u, v \in I_V$ with $\{u, v\} \in I_E$ there exists $i \in \{1,2,\ldots,\delta\}$ such that $\{u, v\} \subseteq I_V^i$; that is, all the edges are covered by complete sub-graphs.

Observe that if no further objective is imposed on the decomposition into complete sub-graphs, it can be easily constructed by setting $\delta = |I_E|$ and putting endpoints of each edge into its own decomposition vertex set. On the other hand, the construction of decomposition with respect to any reasonable objective (such as maximizing the size of complete sub-graphs or minimizing number δ) is a difficult task [15, 22].

In our approach we try to obtain large complete sub-graphs. However, this requirement is not that strict so we have settled for a greedy approach for the construction of decomposition. The greedy algorithm used in our work is shown using a pseudo-code as Algorithm 1 ($\deg_{(V,E)}(v)$ denotes the number of edges from E adjacent to $\in V$).

The algorithm always prefers a vertex with the highest degree with respect to the remaining set of edges. Such a vertex is included into the constructed complete graph and the task is reduced to its neighborhood. This is repeated until the neighborhood of the currently constructed complete sub-graph is empty (a neighborhood of a complete sub-graph is a set of vertices that are connected to all of the vertices of the sub-graph). Once the complete sub-graph is finished its edges are removed from the original graph and the process continues until there are no edges.

The construction of a decomposition as shown in Algorithm 1 heuristically prefers a construction of a large complete sub-graph at the beginning. This strate-

gy proved to produce decompositions of acceptable quality for sub-sequent usage within the B2C-consistency enforcing algorithm.

Proposition 1 (Greedy Time/Space Complexity). A greedy algorithm for the decomposition of a graph interpretation $I(K_{+}^{2}) = (I_V, I_E)$ into complete sub-graphs can be implemented to have the worst case time complexity of $\mathcal{O}(|I_E||I_V|^2)$. The corresponding worst case space complexity is of $\mathcal{O}(|I_V| + |I_E|)$. ■

Commentary: Observe that there may be up to $|I_E|$ complete sub-graphs in the decomposition (each edge constitutes a decomposition set). All the edges of the input graph interpretation may be investigated within the construction of an individual complete sub-graph which adds $|I_E|$ steps (which is $\mathcal{O}(|I_V|^2)$. Adding a vertex with the maximum degree into a complete sub-graph consumes $|I_V|$ steps while it may be repeated up to $|I_V|$ times. Altogether, we have $|I_V|^2$ steps for one complete sub-graph.

Regarding the space complexity it can be argued that several copies of the input graph need to be stored, which makes $\mathcal{O}(|I_V| + |I_E|)$ if the neighborhood of a vertex is represented using linked lists. ■

Algorithm 1. *Greedy algorithm for decomposing a graph interpretation into complete sub-graphs.* The output decomposition is returned as a sequence of decomposition sets of vertices where each of them induces a complete sub-graph.

function *Decompose-Graph-Interpretation*($I(K_{+}^{2}) = (I_V, I_E)$): **sequence**
/* Parameters: $I(K_{+}^{2})$ - a graph interpretation for decomposing */
1: $\delta \leftarrow 1$
2: **while** $I_E \neq \emptyset$ **do**
3: $I_V^{\delta} \leftarrow \emptyset$
4: $(T_V, T_E) \leftarrow (I_V, I_E)$ /* an auxiliary graph for gradual dismantling */
5: **while** $T_V \neq I_V^{\delta}$ **do**
6: **let** $v_{max} \in T_V \setminus I_V^{\delta}$ be a vertex such that $\deg_{(T_V,T_E)}(v_{max}) =$
7: $\max\{\deg_{(T_V,T_E)}(v) \mid v \in T_V \setminus I_V^{\delta}\}$
8: $I_V^{\delta} \leftarrow I_V^{\delta} \cup \{v_{max}\}$
9: $T_V \leftarrow T_V \setminus \{u \mid \{v_{max}, u\} \notin T_E\})$
10: $T_E \leftarrow T_E \cap \binom{T_V}{2}$
11: $I_E \leftarrow I_E \setminus \binom{I_V^{\delta}}{2}$
12: $I_V \leftarrow I_V \setminus \{v \in I_V \mid \deg_{(I_V,I_E)}(v) = 0\}$
12: $\delta \leftarrow \delta + 1$
13: **return** $[I_V^1, I_V^2, \ldots, I_V^{\delta}]$

There are some more properties of the decomposition into complete sub-graphs. Note that decomposition sets intersect vertices corresponding to a domain of a single variable at most once. This is due to the fact that there are no edges between vertices corresponding to a single domain and due to condition (iii). On the other hand, a single vertex may be included in several decomposition sets.

3.3. *B2C-Consistency Enforcing Algorithm*

B2C-consistency will be defined algorithmically as this is the most natural way to do that. Suppose that a decomposition into complete sub-graphs of a given graph interpretation has already been constructed. The basic idea is to enforce bounded $(2, k)$-consistency using only Λ steps in the search for a supporting k-tuple. This search will be accompanied by a special pruning which will use the decomposition into complete sub-graphs to obtain more global reasoning.

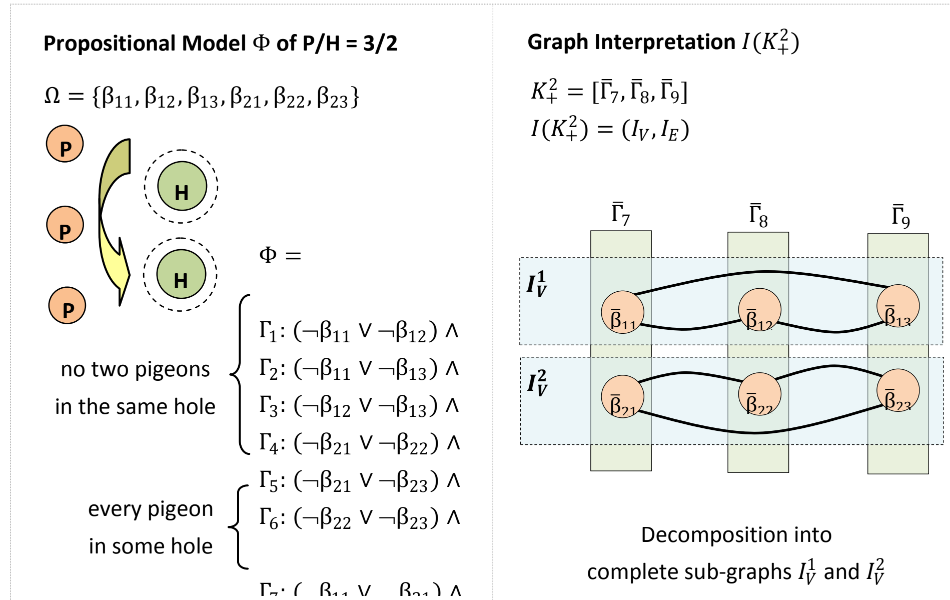


Figure 2. *Pigeon hole (P/H) principle – graph interpretation with complete sub-graphs.* The standard propositional model of the P/H principle Φ for $p = 3$ and $h = 2$ is shown in the left part. A graph interpretation over the explicit literal encoding of Φ with selected variables $\bar{\Gamma}_7$, $\bar{\Gamma}_8$, and $\bar{\Gamma}_9$ is shown in the right part together with its decomposition into complete sub-graphs (notice that the decomposition shown here can be found by the presented greedy algorithm – Algorithm 1).

It is supposed that the search is done in a some systematic way by extending a partial selection of a supporting tuple of values. Regardless of the exact process of the search for the support, we can assume that some values/vertices are selected

into the partial supporting tuple at every step of the process. The selection automatically rules out several other values/vertices – more precisely, values/vertices that are present together with the selected ones in some complete sub-graph are ruled out (this is due to the condition that no more than one literal can be selected in a complete sub-graph).

Algorithm 2. Search for a supporting k-tuple of values within B2C-consistency. It is supposed that a decomposition into complete sub-graphs $\mathcal{J}$ of a given graph interpretation $I(K_+^2)$ with respect to a $(k+2)$-list of variables K_+^2 has already been calculated.

```
function Check-B2C-Consistency(ψ̄_{t_0}^{i_0}, ψ̄_{t_{k+1}}^{i_{k+1}}, I(K_+^2) = (I_V, I_E), 𝒥, Λ): propositional
      /* Parameters: ψ̄_{t_0}^{i_0}, ψ̄_{t_{k+1}}^{i_{k+1}}  - a pair of values for consistency checking
                     I(K_+^2)      - a graph interpretation for decomposing,
                     𝒥             - a decomposition of I(K_+^2) into
                                     complete sub-graphs,
                     Λ             - the number of allowed search steps. */
1:    (ω, Λ) ← Search-B2C-Support(ψ̄_{t_0}^{i_0}, ψ̄_{t_{k+1}}^{i_{k+1}}, ∅, I(K_+^2) = (I_V, I_E), 𝒥, Λ)
2:    return ω

function Search-B2C-Support(ψ̄_{t_0}^{i_0}, ψ̄_{t_{k+1}}^{i_{k+1}}, S, I(K_+^2) = (I_V, I_E), 𝒥, Λ): pair
      /* Parameters: S        - a set of already selected supports. */
1:    if |S| = k then return (TRUE, Λ)
2:    let [ψ̄_{t_1}^{i_1}, ψ̄_{t_2}^{i_2}, ..., ψ̄_{t_l}^{i_l}] = S
3:    for each ψ̄_{t_{l+1}}^{i_{l+1}} ∈ D(Γ̄_{i_{l+1}}) do
4:       if Λ ≤ 0 then return (TRUE, Λ) /* all the steps were consumed */
5:       α ← TRUE
6:       for each I_V ∈ 𝒥 do /* check of constraints */
7:           if |I_V ∩ (∪[ψ̄_{t_0}^{i_0}].S.[ψ̄_{t_{l+1}}^{i_{l+1}}, ψ̄_{t_{k+1}}^{i_{k+1}}])| > 1 then α ← FALSE
8:       let ℒ = {I_V ∈ 𝒥 | I_V ∩ (∪ S.[ψ̄_{t_{l+1}}^{i_{l+1}}]) = ∅}
9:       if |ℒ| + |S| < k then α ← FALSE /* global check */
10:      if α then
11:          if l + 1 < k then /* some supports still remain to be found */
12:              (ω, Λ) ← Search-B2C-Support(ψ̄_{t_0}^{i_0}, ψ̄_{t_{k+1}}^{i_{k+1}}, S.[ψ̄_{t_{l+1}}^{i_{l+1}}],
13:                                           I(K_+^2), 𝒥, Λ)
14:              if ω then return (TRUE, Λ)
15:          else /* all the supports have been found */
16:              return (TRUE, Λ)
17:      Λ ← Λ − 1
18:   return (FALSE, Λ)
```

Nevertheless, the main innovative reasoning mechanism uses the decomposition in a different way. At every point of the process there are still some candidate values/vertices for selection into the final supporting k-tuple. Each one is included in some decomposition sets from which no value/vertex has been selected yet. Let $\mathcal{L}$ be a set of such not yet used decomposition sets and let S be a set of already selected vertices. As only one value/vertex can be selected from each

complete sub-graph we can make the following pruning: if it happens that $|\mathcal{L}| + |S| < k$, the search in the current branch of the support search tree can be terminated as it is not possible to extend the partial selection so that it will finally consist of k elements. This kind of reasoning is especially useful for problems with **non-local** properties such as the P/H principle or FPGA Switch-Box routing [1]. For illustration see Figure 2 (if $\bar{\beta}_{11}$ and $\bar{\beta}_{23}$ have been already selected, then $\mathcal{L} = \emptyset, k = 1$, and $S = \{\bar{\beta}_{11}, \bar{\beta}_{23}\}$ and hence we can conclude that $\bar{\beta}_{11}$ and $\bar{\beta}_{23}$ are inconsistent).

The process of B2C-consistency enfording for a pair of values and a fixed list of variables $K_+^2 = [\bar{\Gamma}_{i_0}, \bar{\Gamma}_{i_1}, \bar{\Gamma}_{i_2}, \ldots, \bar{\Gamma}_{i_k}, \bar{\Gamma}_{i_{k+1}}]$ is shown as Algorithm 2. The algorithm searches for a supporting k-tuple of values for a given pair of values $\bar{\psi}_{t_0}^{i_0} \in D(\bar{\Gamma}_{i_0})$ and $\bar{\psi}_{t_{k+1}}^{i_{k+1}} \in D(\bar{\Gamma}_{i_{k+1}})$ in domains of $\bar{\Gamma}_{i_1}, \bar{\Gamma}_{i_2}, \ldots, \bar{\Gamma}_{i_k}$. The search is done through a systematic extension of the current partial selection of supporting values/vertices. This functionality is implemented using recursive calls, which simulates chronological backtracking search.

The algorithm for enforcing B2C-consistency for a pair of values should be regarded as an incomplete proof of non-existence of a support. That is, if the algorithm finds the given pair of values to be inconsistent then there is actually no support for them (that is, it managed to prove that there is no support using Λ search steps and other techniques; $FALSE$ is returned by *Check-B2C-Consistency* in this case). However, if it does not find the given pair of values to be inconsistent, one of the following cases might happen: a supporting k-tuple of values was found or the algorithm ran out of the allowed number of search steps Λ ($TRUE$ is returned in this case).

Proposition 2 (B2C Time/Space Complexity). If $\Lambda = \infty$ then the algorithm for enforcing B2C-consistency with a decomposition into complete sub-graphs $\mathcal{J}$ of a graph interpretation $I(K_+^2) = (I_V, I_E)$ of a $(k+2)$-list of variables K_+^2 can be implemented to have the worst case time complexity of $\mathcal{O}(k|\mathcal{J}||\mathbb{D}|^{\mathrm{k}})$; otherwise, the worst case time complexity is $\mathcal{O}(|\mathcal{J}|\Lambda)$. The corresponding worst case space complexity is $\mathcal{O}(|I_V| + |I_E|)$. ■

Commentary: It is not difficult to observe that the algorithm needs to go through all the $|\mathbb{D}|^k$ k-tuples in the worst case if the number of the allowed search steps Λ is unbounded. Checking a k-tuple may consume up to $k|\mathcal{J}|$ constraint checks (namely checks against complete sub-graphs). If Λ is bounded then obviously at most Λ steps are done while each step consumes up to $|\mathcal{J}|$ constraint checks.

As all the data elements are accessed sequentially no extra data structures are needed. Hence, we need to store graph interpretation and its decomposition into complete sub-graphs, which we already know to be of $\mathcal{O}(|I_V| + |I_E|)$. The space needed to store the resulting k-tuple is again of $\mathcal{O}(|I_V| + |I_E|)$. ■

Here it depends on our perception of k. It is natural to perceive it as a part of the input and hence the complexity of search for a support is exponential with unbounded Λ. Therefore, the time consumption represents the main bottleneck of the method. However, having the global reasoning based on complete sub-graphs, still much can be done in Λ steps while Λ is bounded.

4. Building a SAT Preprocessing Tool

We intended to use B2C-consistency as a basis for a SAT preprocessing tool. As we have seen, it may not be simply used for that task in its raw form due to its time complexity. A good compromise between the computational effort and strength of the inference has to be found. This section describes how a list of variables should be chosen and how to set particular parameters of B2C-consistency to make it suitable for the intended preprocessing tool.

4.1. *Selection of k-tuples of CSP Variables*

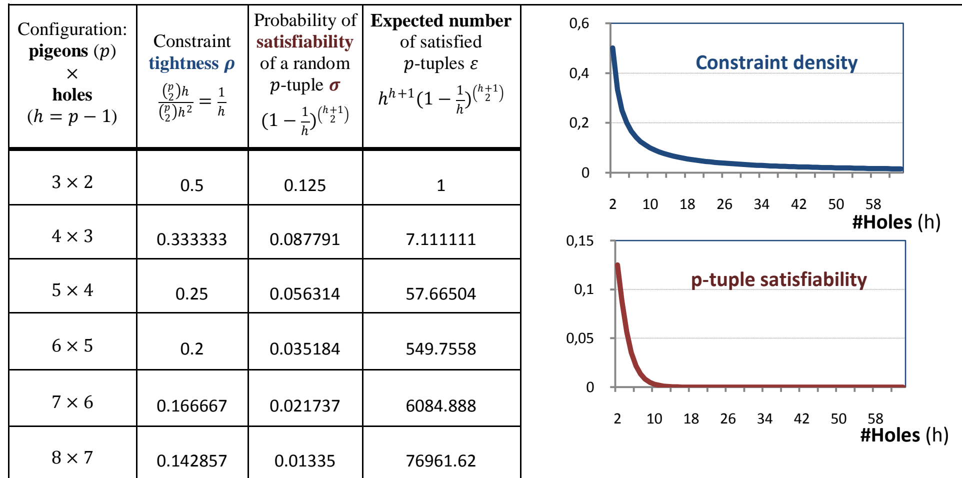
As it is computationally infeasible to achieve B2C-consistency with respect to all the k-tuples of variables and pairs of values in their domains in a non-trivially large SAT instance, some selection of promising subsets of variables on which the consistency will be applied has to be done. The selection is considered to be promising if there is a chance that the consistency rules out some pair of values (that is, the ruled out pair of values cannot be a part of any solution). At the same time, the information captured in the fact that a given pair of values is incompatible should be valuable for a SAT solver in a certain sense. This requirement is imposed by the intention to use B2C-consistency as a preprocessing tool. Hence, the information should not be easily derivable by the SAT solver itself since informing the SAT solver about the inconsistency between a pair of trivially incompatible values is not helpful.

Our approach is to select k-tuples of variables induced by a region of the instance with difficult constraint setup that however can be tackled by the consistency. Such a setup provides a chance to extract valuable information by B2C-consistency. The well known SAT model of the *pigeon hole principle* (P/H principle) – more precisely its explicit literal encoding – is a representative of such a

setup which is well known for resisting from being handled by SAT solvers [1]. All the instances of the P/H principle are unsatisfiable. Having a suitable graph interpretation for the P/H principle as it is shown in Figure 2 (that is, clauses modeling that each pigeon is placed in some hole are selected as a $(k+2)$-tuple of CSP variables for the graph interpretation) we are able to calculate various useful probabilistic characteristics.

Let p be the number of pigeons and let h be the number of holes where it holds that $h = p - 1$. A *constraint tightness* ρ in a binary CSP will be defined as the ratio of the number of allowed pairs of values to the number of all the possible pairs of values. Particularly in the case of the P/H principle it holds that $\rho = \frac{\binom{p}{2}h}{\binom{p}{2}h^2} = \frac{1}{h}$ in the graph interpretation as described above.

Table 1. Probabilistic characteristics of the graph interpretation in the P/H principle.



Configuration: **pigeons** (p) × **holes** ($h = p - 1$)	Constraint **tightness ρ** $\frac{\binom{p}{2}h}{\binom{p}{2}h^2} = \frac{1}{h}$	Probability of **satisfiability** of a random p-tuple $\boldsymbol{\sigma}$ $(1-\frac{1}{h})^{\binom{h+1}{2}}$	**Expected number** of satisfied p-tuples ε $h^{h+1}(1-\frac{1}{h})^{\binom{h+1}{2}}$
3×2	0.5	0.125	1
4×3	0.333333	0.087791	7.111111
5×4	0.25	0.056314	57.66504
6×5	0.2	0.035184	549.7558
7×6	0.166667	0.021737	6084.888
8×7	0.142857	0.01335	76961.62

Another interesting characteristic is the probability of a randomly selected assignment of values to p variables σ calculated from the constraint tightness. It is a reasonable assumption that the satisfaction of individual pairs of values within the assignment is independent of each other. Then it holds for the *probability of satisfaction* of a random p-tuple of values that $\sigma = (1-\rho)^{\binom{p}{2}}$ which is $(1-\frac{1}{h})^{\frac{(h+1)h}{2}}$ in the case of the P/H principle.

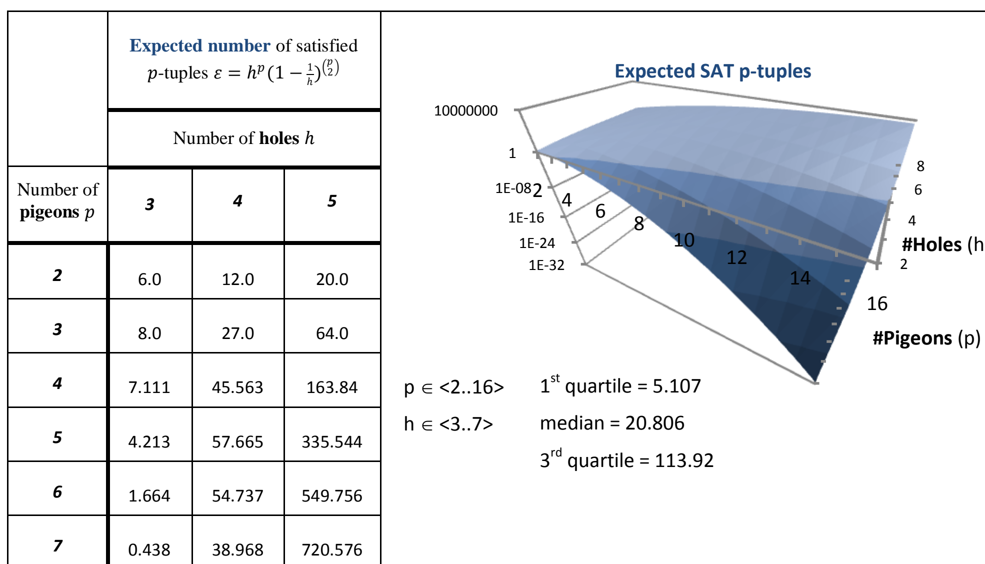
Finally, we will investigate the expected number of satisfied p-tuples of values ε, which will be defined as the total number of possible p-tuples multiplied by ρ. It holds that $\varepsilon = h^p\sigma = h^{h+1}(1-\frac{1}{h})^{\frac{(h+1)h}{2}}$ in the case of the P/H principle. Several examples of probabilistic characteristics are shown in Table 1. The limit beha-

vior of the above characteristics with $h \to \infty$ is summarized in the following easy-to-prove proposition.

Proposition 3 (Limit P/H Characteristics). The probability of satisfiability of a random p-tuple of values ρ in a graph interpretation of the P/H principle converges to 0 for h → ∞; that is, $\lim_{h\to\infty}(1-\frac{1}{h})^{\binom{h+1}{2}} = 0$. The expected number of satisfied p-tuples of values ε in the P/H principle is $\mathcal{O}(e^{(h+1)(-\frac{1}{2}+\ln h)})$ which is $\mathcal{O}\left(\left(\frac{h}{\sqrt{e}}\right)^{(h+1)}\right)$ and blows up to $+\infty$ for $h \to \infty$; that is, $\lim_{h\to\infty} h^{h+1}(1-\frac{1}{h})^{\binom{h+1}{2}} = +\infty$. ■

We will generalize the P/H principle so that there will be strictly less holes than pigeons but not necessarily one fewer. The generalized P/H principle is unsatisfiable as well. A sample of probabilistic characteristics of the model of the generalized P/H principle is shown in Table 2.

Table 2. Expected number of satisfied tuples of values in the generalized P/H principle.



Number of pigeons p	Expected number of satisfied p-tuples $\varepsilon = h^p(1-\frac{1}{h})^{\binom{p}{2}}$ Number of **holes** h: 3	4	5
2	6.0	12.0	20.0
3	8.0	27.0	64.0
4	7.111	45.563	163.84
5	4.213	57.665	335.544
6	1.664	54.737	549.756
7	0.438	38.968	720.576

Our aim is to select $(k+2)$-tuples of CSP variables for B2C-consistency in the explicit literal encoding $E^1(\Phi) = (X^1_\Phi, D^1_\Phi, C^1_\Phi)$ which has similar probabilistic characteristics that are exhibited by the model of the (generalized) P/H principle. This selection is supposed to ensure the required properties – that is, a simi-

lar level of difficulty as the P/H principle and the similar constraint setup. The following incremental mechanism for selecting the next variable based on estimating probabilistic characteristics from the currently selected variables will be used.

The requirement which is specified as a part of the input together with k is the interval for the expected number of satisfied $(k+2)$-tuples of values. Let ε_L and ε_U be the lower and upper bound for this interval respectively. The first CSP variable into the $(k+2)$-tuple is supposed to be selected using some specific process (randomly or systematically; actually a systematic process is used within the experimental implementation). Other CSP variables are selected incrementally; suppose that $K_+^2 = [\bar{\Gamma}_{i_0}, \bar{\Gamma}_{i_1}, \bar{\Gamma}_{i_2}, \dots, \bar{\Gamma}_{i_\kappa}]$ is a tuple of the already selected CSP variables (if $\kappa = k$ then the process is finished). Let $\bar{\Gamma}_{i_{\kappa+1}}$ be a candidate CSP variable.

Algorithm 3. *Process of selecting a suitable* $(k+2)$*-tuple of CSP variables.* Variables are heuristically selected to prefer the resulting expected number of satisfied $(k+2)$-tuples of values in the interval of $\langle\varepsilon_L, \varepsilon_U\rangle$ or near this interval from below or above.

---

function *Select-CSP-Variables*$(k, \bar{\Gamma}_{i_0}, E^1(\Phi) = (X_\Phi^1, D_\Phi^1, C_\Phi^1), \varepsilon_L, \varepsilon_U)$: **tuple**

/* Parameters: k - size of the tuple of CSP variables,
$\bar{\Gamma}_{i_0}$ - the first CSP variable,
$E^1(\Phi)$ - explicit literal encoding,
$\varepsilon_L, \varepsilon_U$ - lower and upper bounds for the expected number of satisfied $(k+2)$-tuples of values. */

1: **for** $\kappa = 0,1,\dots,k$ **do**
2: **for each** $\bar{\Gamma}_{i_{\kappa+1}} \in X_\Phi^1$ **do**
3: **let** $\rho(\bar{\Gamma}_{i_{\kappa+1}})$ is the constraint tightness in $\bigcup[\bar{\Gamma}_{i_0}, \bar{\Gamma}_{i_1}, \bar{\Gamma}_{i_2}, \dots, \bar{\Gamma}_{i_{\kappa+1}}]$
4: $\varepsilon(\bar{\Gamma}_{i_{\kappa+1}}) \leftarrow \left(\sqrt[\kappa+1]{\prod_{l=0}^{\kappa+1} |D(\bar{\Gamma}_{i_l})|}\right)^{k+2} \left(1-\rho(\bar{\Gamma}_{i_{\kappa+1}})\right)^{\binom{k+2}{2}}$
/* the following let form assigns $\perp$ if undefined */
5: **let** $\bar{\Gamma}_\alpha \in X_\Phi^1$ such that $\varepsilon(\bar{\Gamma}_\alpha) \leq \varepsilon_L \wedge (\forall \bar{\Gamma}_l \in X_\Phi^1)\varepsilon(\bar{\Gamma}_\alpha) \geq \varepsilon(\bar{\Gamma}_l)$
6: **let** $\bar{\Gamma}_\beta \in X_\Phi^1$ such that $\varepsilon(\bar{\Gamma}_\beta) \geq \varepsilon_U \wedge (\forall \bar{\Gamma}_l \in X_\Phi^1)\varepsilon(\bar{\Gamma}_\beta) \leq \varepsilon(\bar{\Gamma}_l)$
7: **let** $\bar{\Gamma}_\mu \in X_\Phi^1$ such that $\varepsilon_L \leq \varepsilon(\bar{\Gamma}_\mu) \leq \varepsilon_U$
8: **if** $\bar{\Gamma}_\mu \neq \perp$ **then**
9: $i_\kappa \leftarrow \mu$
10: **else**
11: **if** $\bar{\Gamma}_\alpha = \perp$ **then**
12: $i_\kappa \leftarrow \beta$
13: **else**
14: **if** $|\varepsilon(\bar{\Gamma}_\alpha) - \varepsilon_L| < |\varepsilon(\bar{\Gamma}_\beta) - \varepsilon_U|$ **then**
15: $i_\kappa \leftarrow \alpha$
16: **else**
17: $i_\kappa \leftarrow \beta$
18: **return** $[\bar{\Gamma}_{i_0}, \bar{\Gamma}_{i_1}, \bar{\Gamma}_{i_2}, \dots, \bar{\Gamma}_{i_\kappa}]$

The expected number of the satisfied $(k+2)$-tuples with $\bar{\Gamma}_{i_{\kappa+1}}$ denoted as $\varepsilon(\bar{\Gamma}_{i_{\kappa+1}})$ is estimated as follows: let $\rho(\bar{\Gamma}_{i_{\kappa+1}})$ be the constraint tightness among

variables from the set $\bigcup K_{+}^{2} \cup \{\bar{\Gamma}_{i_{\kappa+1}}\}$ (already selected variables together with the new candidate) then $\varepsilon(\bar{\Gamma}_{i_{\kappa+1}}) = \left(\sqrt[\kappa+1]{\prod_{l=0}^{\kappa+1}|D(\bar{\Gamma}_{i_l})|}\right)^{k+2} \left(1-\rho(\bar{\Gamma}_{i_{\kappa+1}})\right)^{\binom{k+2}{2}}$. That is, the product of sizes of domains of the final $(k+2)$-tuple is estimated as $(k+2)$th power of the geometric mean of sizes of the domain of already selected variables. The constraint tightness is supposed to be preserved for the final $(k+2)$-tuple. If $\varepsilon_L \leq \varepsilon(\bar{\Gamma}_{i_{\kappa+1}}) \leq \varepsilon_U$ then $\bar{\Gamma}_{i_{\kappa+1}}$ may be used as the next CSP variable for the $(k+2)$-tuple. If there are multiple variables satisfying this condition any of them may be selected (in the implementation that one with ε closest to $\frac{\varepsilon_L+\varepsilon_U}{2}$ is selected). The whole process of selection of CSP variables for B2C-consistency is formalized as Algorithm 3.

Proposition 4 (Selection Time/Space Complexity). The algorithm for selecting CSP variables can be implemented to have the worst case time complexity of $\mathcal{O}(k^2|X_{\Phi}^1||\mathbb{D}|^2)$. A space of $\mathcal{O}(k+|X_{\Phi}^1|)$ is needed in addition to the space necessary for storing CSP $E^1(\Phi)$. ■

Commentary: Each new CSP variable is selected for the resulting tuple out of at most $|X_{\Phi}^1|$ CSP variables for which estimation of the expected number of satisfied $(k+2)$-tuples must be calculated. Calculating this estimation with respect to a single variable consumes $\mathcal{O}(k|\mathbb{D}|^2)$ steps as it is necessary to calculate constraint tightness relatively to all the already selected variables. A new variable is included exactly k times.

Additional space is needed for storing probabilistic characteristics for CSP variables, which consumes the space of $\mathcal{O}(|X_{\Phi}^1|)$. A space of $\mathcal{O}(k)$ is needed to store the resulting tuple of CSP variables. ■

It is infeasible in large SAT instances to compute and to store constraint tightness between all the pairs of variables on the current commodity hardware because there are too many such pairs (notice that there may be more than $1.0\text{E}+6$ clauses in large SAT instances which makes more than $\binom{1.0\text{E}+6}{2} \approx 1.0\text{E}+12$ pairs of variables; that would require approximately several terabytes of memory). Hence, it is necessary to compute constraint tightness on demand.

4.2. *SAT Preprocessing with B2C-Consistency*

An experimental SAT preprocessing tool based on B2C-consistency called `preprocessSIGMA` [27] was implemented in C++ in order to conduct an experimental evaluation and to provide proof of the concept. To achieve the best infe-

rence strength of preprocessing, $(k + 2)$-tuples are selected according to the theory in the previous section so that the expected number of satisfied tuples of values belongs into the interval typical for the model of the generalized P/H principle. We select k uniformly from the interval $\langle 2..10 \rangle$ as it experimentally proved to be computationally manageable in reasonable time.

In typical SAT instances arity of clauses ranges from 2 to 10 [18] while the most common are small clauses with arities 3, 4, and 5 – domain sizes in the corresponding literal encoding are exactly the same. The expected number of satisfied tuples of values for a setup of the P/H principle with corresponding $p \in \langle 2..10 \rangle$ and $h \in \langle 3..5 \rangle$ belongs into the interval $\langle 0.001,755.579 \rangle$ while the 1st quartile, median, and 3rd quartile are equal to 5.107, 20.806, 113.92, respectively. Taking into account that we are preferring the non-existence of satisfied tuple of values, it is advisable to select the preferred interval for the expected number of satisfied tuples of values $\langle \varepsilon_L, \varepsilon_U \rangle$ with ε_L low below the median and slightly below the 1st quartile and ε_U slightly above the median. A preliminary experimental evaluation with SAT instances containing mainly small clauses showed that the best setting is $\langle \varepsilon_L, \varepsilon_U \rangle = \langle 3.0,32.0 \rangle$ which well correlates with the above probabilistic estimations. The use of different bounds resulted in deriving less valuable forbidden pairs of values in the preprocessing step (that is, explicit forbidding of such pairs by adding new clauses had a limited positive effect).

As the computation of B2C-consistency is a time consuming operation it is done only for a certain number of tuples of variables. More precisely, small formulae with less than or equal to 2048 variables are allowed 16 times the number variables B2C-consistency checks. Large formulae (that is, those with more than 2048 variables) are allowed 4 times square root of the number of variables B2C-consistency checks (currently, there is no smooth transition between these two rates as it was not necessary to be implemented for experimental evaluation). In both cases, the number of steps of the search for a consistent k-tuple was bounded by the constant $\Lambda = 4096$. This setup of Λ was manually tailored during the development of the method.

We are aware that the presence of several parameters in the method may be problematic since the user is required to set them. However, in our analysis we provide some ideas for their setting and, most importantly, the parameters can be regarded as an opportunity for further optimization by methods for automated parameter tuning (*programming by optimization*) [17].

5. Experimental Evaluation

The experimental evaluation of our prototype SAT preprocessor preprocess-SIGMA focused on discovering the benefit of B2C-consistency in the context of other existent preprocessing techniques and on the evaluation of internal properties of the experimental implementation. It also should provide a justification for the theory we have discussed earlier.

5.1. *Basic Competitive Experimental Evaluation*

The experimental implementation of B2C-consistency within our prototype tool preprocessSIGMA has been competitively evaluated with respect to the most prominent existing tools for SAT preprocessing. Particularly, the following preprocessing tools have been evaluated: LiVer [25], NiVer [25], HyPre [5], Shatter with Saucy version 3.0 [2, 21] (here abbreviated as saucy-3), and the technique of *blocked clause elimination* [16, 20] (here abbreviated as BCE) implemented within precosat-465 [20] (here abbreviated as BCE). As the reference SAT solver MiniSAT version 2.2 [13] with an built-in SatElite preprocessing step has been used.

LiVer and NiVer use resolution-based variable elimination for preprocessing; LiVer allows a bounded increase in the total number of literals in the resulting formula while NiVer does not allow any increase in this number. The HyPre preprocessing tool is based on binary hyper-resolution and equivalence reasoning. Shatter represents a tool most akin to our preprocessSIGMA as it employs a certain kind of global reasoning as well. Symmetries in the input formula are detected and symmetry-breaking clauses are added by Shatter into the output formula. To detect symmetries, the *graph isomorphism* problem [28] needs to be solved during the preprocessing process which is done by the Saucy module. The performance of the Saucy module is crucial in Shatter.

The experimental evaluation was done with a set of 344 difficult SAT instances (mixture of satisfiable and unsatisfiable) taken from the *Satisfiability Library* (SATLib – only structured instances have been taken) [18] and from the crafted category of the *SAT Competitions 2002/2003* and *2007/2009* (all the problems from the crafted category of a size up to 600kB have been taken). The complete set of instances used in the experimental evaluation can be found at the website: http://ktiml.mff.cuni.cz/~surynek/research/j-preprocess-2011. This website also contains experimental data in the raw form and the complete source code in C++ necessary to reproduce all the presented experiments.

Table 3. *Listing of results for a fraction of the set of testing instances used in our experimental evaluation.* The number of conflicts `MiniSAT 2.2` encountered on the original instances and on those preprocessed by `HyPre`, `LiVer`, `NiVer`, `saucy-3`, and our `preprocessSIGMA` are shown. The best performing preprocessors on each instance are marked in bold (the timeout for both preprocessing and `MiniSAT` was set to 256 seconds). Observe the large differences among individual preprocessors.

Conflicts	***Variables***	***Clauses***	***C/V Ratio***	***Original***	***HyPre***	***BCE***	***LiVer***	***NiVer***	***saucy-3***	***sigma***	***SAT/UNSAT***
bart12.shuffled	*180*	*820*	4.555	**105**	212	**105**	118	118	603	**105**	SAT
bart14.shuffled	*195*	*905*	4.641	104	402	104	**100**	**100**	102	104	SAT
bart16.shuffled	*210*	*990*	4.714	**103**	106	**103**	**103**	**103**	215	**103**	SAT
bart20.shuffled	*270*	*1476*	5.466	121	127	206	**103**	**103**	160	121	SAT
ca004.shuffled	*80*	*168*	2.1	43	**29**	48	32	**29**	42	43	UNSAT
ca008.shuffled	*130*	*370*	2.846	145	117	175	**102**	150	151	145	UNSAT
ca016.shuffled	*272*	*780*	2.867	449	**293**	465	416	326	357	433	UNSAT
ca032.shuffled	*558*	*1606*	2.878	943	752	1103	739	**657**	901	790	UNSAT
difp_19_99_arr_rcr	*1201*	*6563*	5.464	209417	141649	343814	58305	304092	209417	**92754**	SAT
difp_19_99_wal_rcr	*1775*	*10446*	5.885	134284	**31031**	92343	108681	158235	N/A	15245	SAT
difp_21_1_arr_rcr	*1453*	*7967*	5.483	191884	63546	126655	538426	427292	191884	**45453**	SAT
difp_21_99_arr_rcr	*1453*	*7967*	5.483	190663	97408	66191	249983	350142	190663	**35704**	SAT
dp04u03.shuffled	*1017*	*2411*	2.370	70	**26**	77	72	63	N/A	61	UNSAT
dp05s05.shuffled	*1885*	*4818*	2.555	90	138	80	116	100	N/A	**46**	SAT
ezfact32_6.shuffled	*769*	*4777*	6.211	422	33088	**169**	32957	32957	422	209	SAT
ezfact32_7.shuffled	*769*	*4777*	6.211	5744	29574	**173**	46659	46659	5744	836	SAT
ezfact32_9.shuffled	*769*	*4777*	6.211	1181	47191	218	64056	64056	1181	**160**	SAT
ezfact32_10.shuffled	*769*	*4777*	6.211	1990	1988	**406**	22500	22500	1990	448	SAT
fpga10_11_uns_rcr	*220*	*1122*	5.1	4935017	8315862	4935017	4866421	4866421	548002	**2**	UNSAT
fpga10_12_uns_rcr	*240*	*1344*	5.6	7209341	7219129	7140410	7183640	7218248	645603	**1**	UNSAT
fpga10_13_uns_rcr	*260*	*1586*	6.1	6466487	6511919	5963904	6497147	6497268	264637	**1**	UNSAT
fpga10_15_uns_rcr	*300*	*2130*	7.1	5390760	5401469	5361172	5387934	5405715	91837	**1**	UNSAT
fpga10_8_sat	*120*	*448*	3.733	201	163	201	201	201	**65**	201	SAT
fpga10_9_sat	*135*	*549*	4.066	202	168	202	202	202	**100**	202	SAT
fpga12_11_sat	*198*	*968*	4.888	200	405	200	200	200	**54**	200	SAT
fpga12_12_sat	*216*	*1128*	5.222	208	102	208	208	208	**36**	208	SAT
homer06.shuffled	*180*	*830*	4.611	272019	209811	272019	258487	258487	39341	**1**	UNSAT
homer10.shuffled	*360*	*3460*	9.611	641132	502279	641132	464639	464639	144	**2**	UNSAT
homer16.shuffled	*264*	*1476*	5.590	6525641	6527180	6484372	6766937	6682636	3195152	**3**	UNSAT
homer20.shuffled	*440*	*4220*	9.590	3230156	3249156	3043099	3265756	3207038	87585	**2**	UNSAT
lisa19_0_a.shuffled	*1201*	*6563*	5.464	235824	117828	209804	381242	108878	235824	**15534**	SAT
lisa19_1_a.shuffled	*1201*	*6563*	5.464	445563	439709	688143	**208567**	528589	445563	320076	SAT
lisa21_1_a.shuffled	*1453*	*7967*	5.483	328846	121498	67873	**4841**	309122	328846	93629	SAT
med11.shuffled	*341*	*5556*	16.293	**41**	197	102	101	101	**41**	**41**	SAT
med17.shuffled	*782*	*18616*	23.805	**106**	151	4599	808	808	**106**	**106**	SAT
qg1-7.shuffled	*686*	*6816*	9.935	**49**	115	**44**	67	67	242	49	SAT
term1_gr_2pin_w3.shuffled	*746*	*3517*	4.714	52	69	27	21	124	52	**9**	UNSAT
term1_gr_rcs_w3.shuffled	*606*	*2518*	4.155	7	7	7	7	7	11	**1**	UNSAT

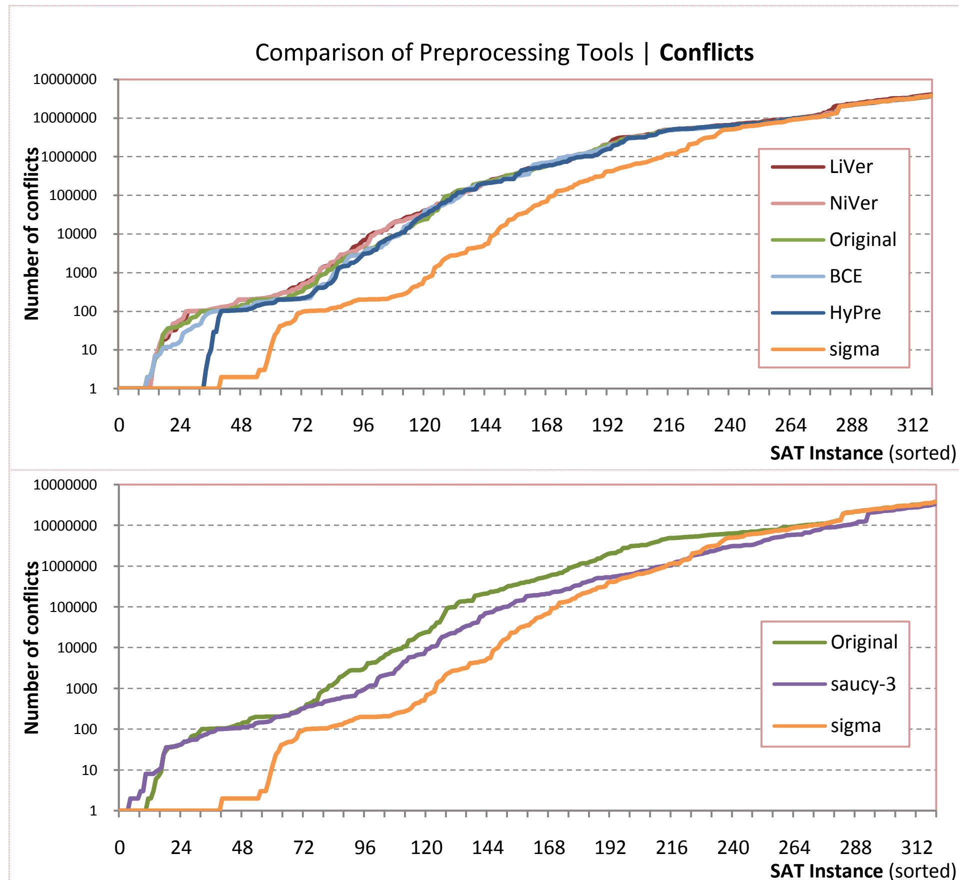


Figure 3. *Competitive comparison of preprocessing tools (conflicts).* The number of conflicts that occurred when solving the original and preprocessed SAT instances by `MiniSAT 2.2` are shown (instances are sorted for each preprocessor to get increasing sequences – easier instances tend to be on the left while hard instances tend to be on the right). Our `preprocessSIGMA` is compared with `HyPre`, `LiVer`, `NiVer`, `BCE`, and `saucy-3` on a set of SAT instances from SATLib and SAT Competitions 2003/2004 and 2007/2009. It can be observed that `HyPre`, `LiVer`, `NiVer`, and `BCE` have only marginal effect (upper part) compared to `preprocessSIGMA` and `saucy-3` (lower part) which both deliver significant improvements. If timeout was reached the instance was excluded from the figure.

Several characteristics were measured during the evaluation process. The most informative characteristic is the number of *conflicts* that occurred during the process of solving. The conflicts can be regarded as a dead-end in the backtracking-based search process. The number of conflicts has been measured for the original instances and for instances processed by individual SAT preprocessors from our test suite. The number of conflicts corresponds well with the overall runtime.

The CPU time* has been measured as well to obtain the complete picture of performance of all the SAT preprocessors.

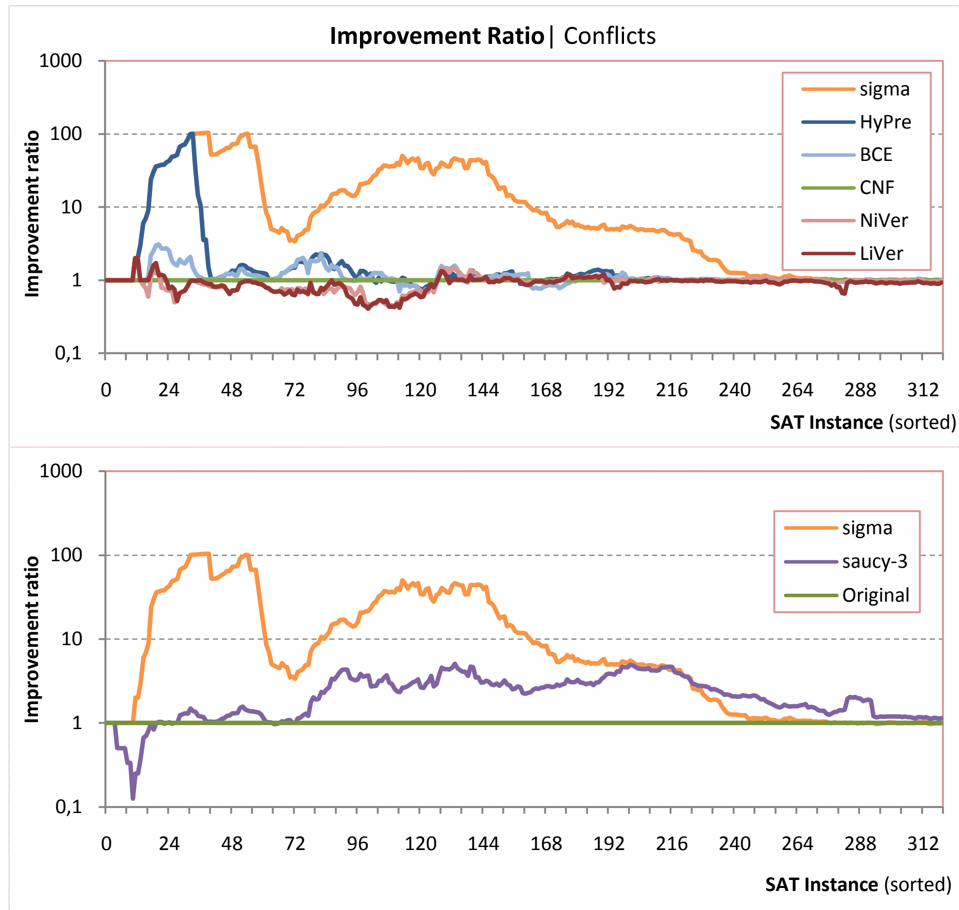


Figure 4. *Improvement ratio in the number of conflicts gained by the application of preprocessor.* The ordering of instances per preprocessor is the same as in Figure 3. It can be observed that `NiVer` and `LiVer` cause worsening in a significant number of instances. `HyPre` is particularly successful on easier instances. The best improvement can be achieved by `saucy-3` and `preprocessSIGMA` while `saucy-3` has an advantage in large instances and `preprocessSIGMA` dominates in easier instances.

A small fraction of the set of instances (38 out of 344) used in the experimental evaluation together with their characteristics and results regarding the number

* All the tests were run on a machine with Intel Xeon 2.0GHz CPU, 12 GB of RAM, under Ubuntu Linux version 8.04, Kernel 2.6.24-19 SMP.

of conflicts after preprocessing is shown in Table 3. In all the tests presented in this paper, preprocessing and solving by `MiniSAT` was run for at most 256 seconds of CPU time; that is, the total runtime per instance is limited to 512 seconds of CPU time.

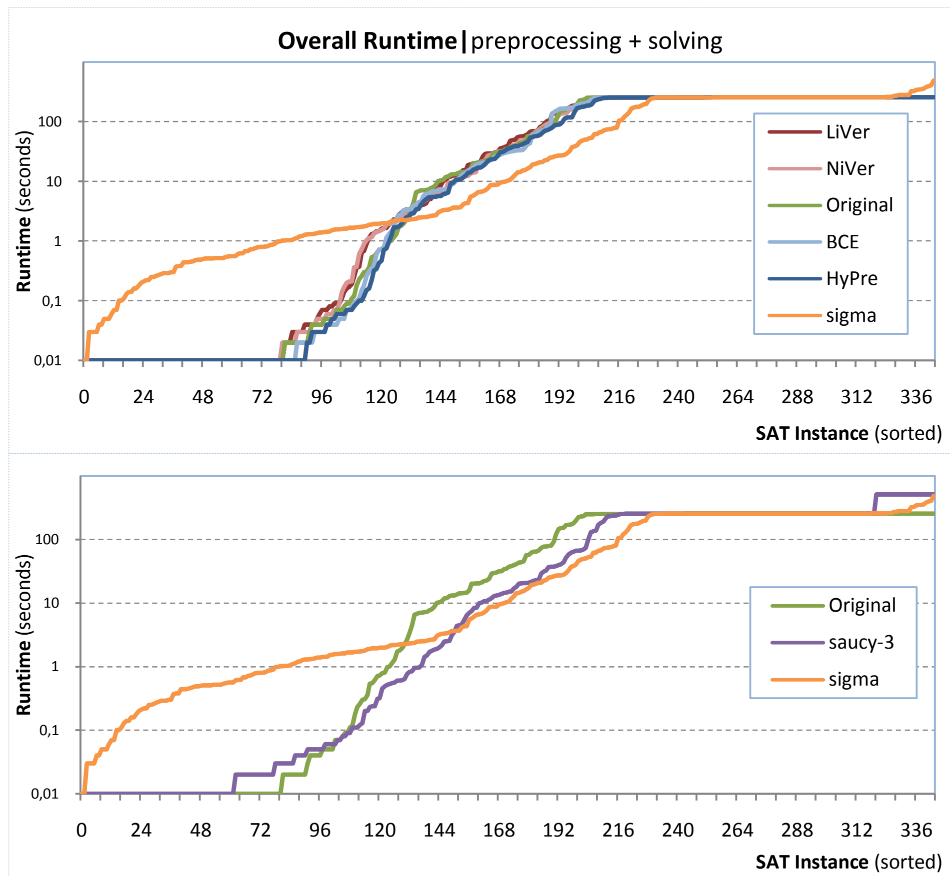


Figure 5. *Competitive comparison of preprocessing tools according to runtime*[†]. The runtime of preprocessing plus the runtime of solving preprocessed instances are shown. Instances are sorted according to the increasing runtime (each preprocessor has its own sorting of instances). The advantage of `preprocessSIGMA` and `saucy-3` is slightly reduced as they both require longer runtime for preprocessing than other preprocessors. However, they are still dominant on medium hard to very hard instances. On easier instances `saucy-3` prevails over `preprocessSIGMA` but the difference is narrowing towards harder instances where `saucy-3` often did not finish in the given timeout and `preprocessSIGMA` became better option.

[†] Notice there are more instances in runtime figures. This is due to the fact that instances where preprocessing did not finish are included in runtime figures but they are not included in figures regarding conflicts.

The full competitive comparison of the number of conflicts that MiniSAT encountered when solving the original instances and preprocessed ones is shown in Figure 3.

There are large improvements and large differences among individual SAT preprocessors observable in Table 3 and Figure 3. Hence, preprocessing seems to be a powerful tool and the choice of the right preprocessor is a crucial decision point with an important performance impact.

The evaluation implies that preprocessors solely relying on simplification through local inferences such as resolution, hyper-resolution, and blocked clause elimination – that is HyPre, LiVer, Niver, and BCE – deliver almost no improvement on the evaluated set of difficult SAT instances (even worsening in a significant number of instances appeared). These results indicate that preprocessing employing local inference rules only is unable to discover and exploit a higher level structure encoded in the instance.

On the other hand, saucy-3 as well as preprocessSIGMA, which both employ global reasoning, deliver significant improvements in terms of the number of conflicts on preprocessed instances. Hence, global reasoning seems to be beneficial in instances encoding a certain kind of a high level structure.

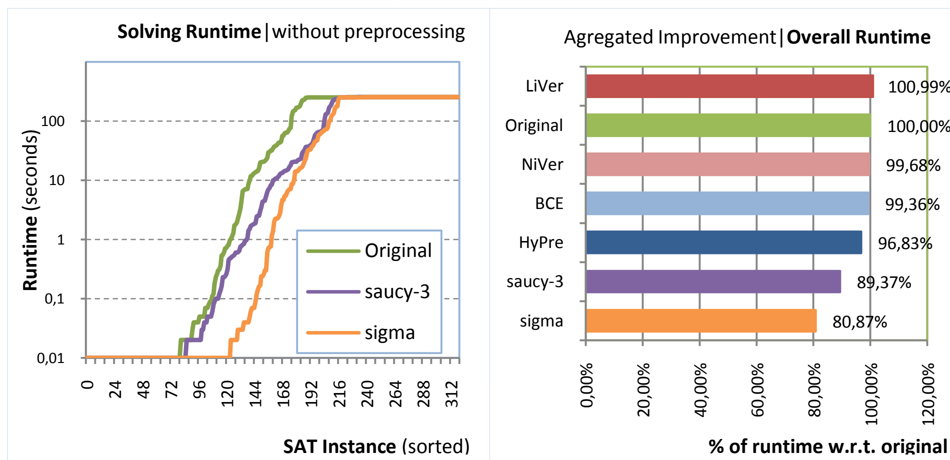


Figure 6. *Additional results regarding runtime – runtime without preprocessing/aggregated improvement.*
Left: If merely the SAT solving runtime is accounted then preprocessSIGMA delivers better performance in easier to moderately difficult instances than saucy-3. In difficult instances the performances of preprocessSIGMA and saucy-3 are matched.
Right: The improvement of the overall solving time over the whole evaluated set of instances. Only preprocessors based on global reasoning – saucy-3 and preprocessSIGMA – deliver considerable improvement. Approximately 20% of the original runtime is saved in the case of preprocessSIGMA.

In instances of easy to medium difficulty, preprocessSIGMA delivers a better positive effect in preprocessing than saucy-3 – up to 100 times less conflicts are encountered in instances preprocessed with preprocessSIGMA than in the original ones. The difference between preprocessSIGMA and saucy-3 diminishes in instances of top difficulty (saucy-3 becomes marginally better in several instances).

The results, however, should not be interpreted as that preprocessing by resolution/hyper-resolution is useless. In simpler instances it is typically more beneficial [4] if we take into account a tradeoff between the benefit and computational costs. Moreover, we need to consider that the version of MiniSAT we used has its own built-in preprocessor SatElite. The results may thus show that simple resolution-based preprocessing is not enough to outperform the benefit of the use of SatElite (although this claim may require further investigation).

An experimental evaluation regarding the runtime is shown in Figure 5 and Figure 6. It can be observed that if merely solving runtime is measured, then the picture is almost the same is in the case of conflicts – preprocessSIGMA and saucy-3 clearly outperforms the others (BCE, HyPre, LiVer, and Niver). The situation changes if the time for preprocessing is accounted (that is, total runtime = preprocessing runtime + solving runtime is taken into account). Here preprocessSIGMA starts lagging behind all others in easier instances due to its long runtime.

A similar phenomenon but not that profound can be observed for saucy-3, which loses against BCE, HyPre, LiVer, and NiVer in easier instances. The situation changes in more difficult instances where saucy-3 and preprocessSIGMA perform better than others. Even preprocessSIGMA matches saucy-3 on yet more difficult instances.

If the total runtime for the whole testing suite is considered, we get an interesting comparison: both saucy-3 and preprocessSIGMA save up to 20% of the total runtime compared to the situation without preprocessing while the other tools (BCE, HyPre, LiVer, and Niver) provide no or marginal improvement only.

Note that the match in overall runtime with saucy-3 in more difficult instances has been achieved despite the not well optimized implementation of preprocessSIGMA (this is also the reason why we need to limit the size of the tested instances). Regarding the preprocessing time with preprocessSIGMA there is a great potential for further improvement.

5.2. *B2C-Consistency on Integer Factorization*

An especially good performance was exhibited by our preprocessing tool based on B2C-consistency in instances encoding *integer factorization problem* [3] (satisfiable instances). The first observation made in these instances is that B2C-consistency is able to make many inferences of inconsistent pairs of values that can be ruled out in the preprocessed instance afterwards.

An additional experimental evaluation showed that the more inconsistent pairs of values are inferred, the greater the reduction of the number of conflicts (as well as runtime) can be achieved on the resulting instance. However, this property contradicts the requirement of bounding the number of B2C-consistency checks which is needed to be low to preserve reasonable time consumption (if we want to infer as many inconsistent pairs of values as possible we should perform as many consistency checks as possible). Hence, there is still room for improvement on integer factorization problems using fine tuning of the parameters of B2C-consistency such as the allowed number of constraint checks.

The competitive results regarding the integer factorization problem are shown in Figure 7. Clearly, `preprocessSIGMA` is the best for almost all the instances in terms of the number of conflicts it can save. Surprisingly, `saucy-3` did not finish preprocessing for approximately half of the instances in the given timeout of 256 seconds. Regarding relative improvement, it rarely happens that the tested preprocessors cause worsening (only `LiVer` and `NiVer` exhibited this behavior marginally).

If we look at the overall runtime, `saucy-3` loses due to its frequent depleting the timeout. Another observation is that accounting preprocessing time does not change the picture of relative performance so much as the solving time for the instances is quite long compared to the preprocessing time.

The cumulative runtime improvement achieved by `preprocessSIGMA` on integer factorization instances is 27.84% compared to 19.13% on the complete set of testing instances.

A surprising result has been obtained for `saucy-3` which was unexpectedly outperformed by all the local inference based preprocessors `BCE`, `HyPre`, `LiVer`, and `NiVer`.

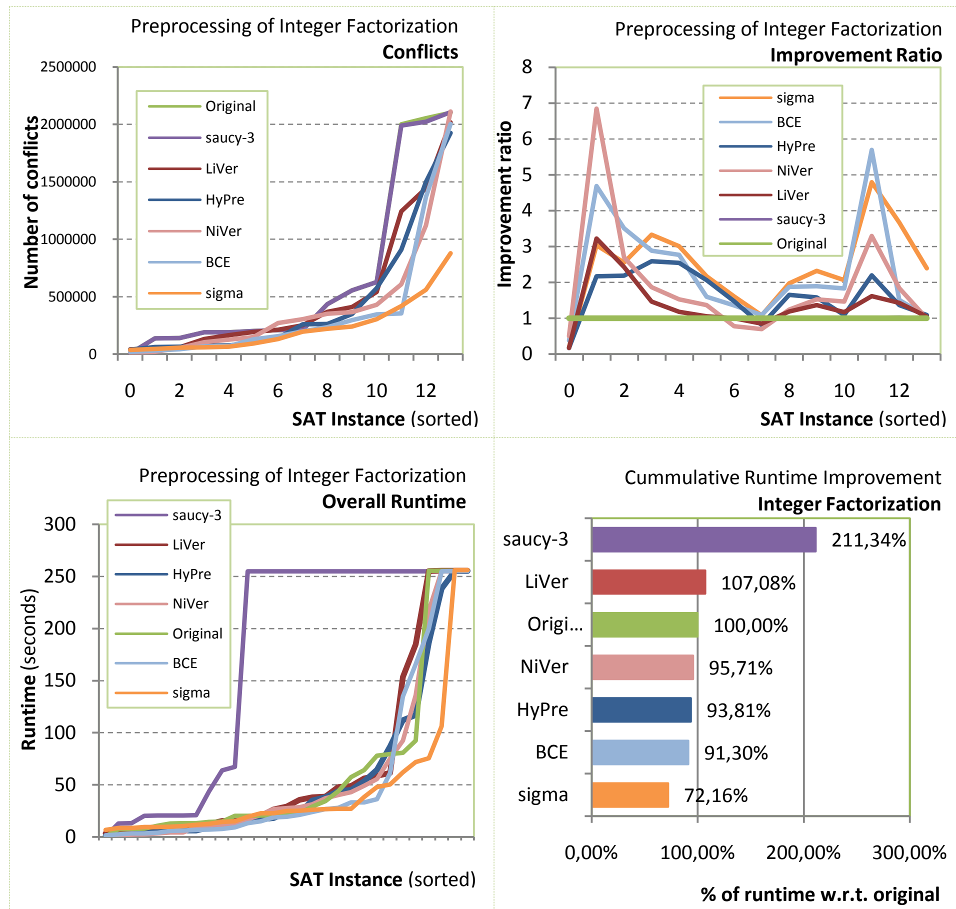


Figure 7. *Competitive evaluation in an instance encoding the integer factorization problem.*
Upper left: The absolute number of conflicts that `MiniSAT 2.2` has encountered in instances encoding integer factorization [*3*] after preprocessing by tested SAT preprocessors is shown. Clearly, `preprocessSIGMA` provides the best performance while `saucy-3` surprisingly lost to all the preprocessors. `BCE` seems to be a good option on integer factorization although it delivers mediocre performance on other instances from the tested set.
Upper right: Improvement ratio in terms of the number of conflicts is shown. Instances are sorted in the same order as in the previous figure.
Lower left: Runtime measurement also includes instances where `saucy-3` did not finish in the given timeout of 256.0 seconds which is approximately half of the instances encoding integer factorization.
Lower right: The aggregated improvement achieved by `preprocessSIGMA` in the overall runtime is $100.00 - 72.16 = 27.84\%$ on integer factorization. The second best `BCE` lost by a significant margin of almost 20% to the winner.

5.3. *Experimental Evaluation of the Variables Selection Process*

The last part of the experiments was devoted to an evaluation of the selection of variables for consistency checks. This evaluation is important in order to verify whether all the internal processes of B2C-consistency worked as expected. This aspect concerns mainly the selection of a list of variables for the consistency check.

The expected number of satisfied tuples of values over the variables selected by Algorithm 3 with the setup of $\langle \varepsilon_L, \varepsilon_U \rangle = \langle 3.0, 32.0 \rangle$ over all the consistency checks on the tested instances has the following probabilistic characteristics – minimum, first quartile, median, third quartile, maximum equal to 2.131, 14.899, 27.562, 130.149, and 1,141,710.567 respectively (in this test only instances from SATLib were used). A more detailed insight into the distribution of the expected number of satisfied tuples of values over selected variables is provided in a partial histogram shown in Figure 8.

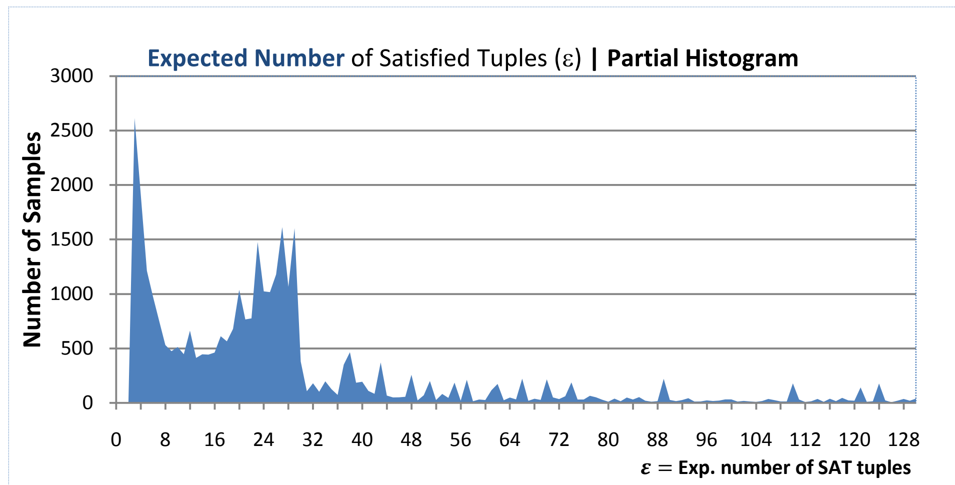


Figure 8. *Partial histogram of the expected number of satisfied tuples (ε).* The histogram characterizes the selection of variables made by Algorithm 3 over all the testing SAT instances and all the B2C-Consistency checks. Only the part up to the 3rd quartile is shown. It can be observed that most of the selections of tuples of variables have the expected number of satisfied tuples of values within the interval $\langle \varepsilon_L, \varepsilon_U \rangle = \langle 3.0, 32.0 \rangle$ as it was required.

5.4. *Summary of Experimental Evaluation*

If we summarize the results of the experimental evaluation we can state that B2C-consistency with the proposed process for the selection of variables represents a powerful technique that can be used as a basis of a SAT preprocessing tool. Our experimental evaluation has proven that prototype preprocessing tool `preprocessSIGMA` based on B2C-consistency is fully competitive with respect to the existent prominent SAT preprocessing tools in terms of saving the number of conflicts as well as in terms of the overall runtime. Competitiveness in terms of runtime has been achieved despite the not well optimized implementation of the prototype.

An especially good performance was exhibited by `preprocessSIGMA` in instances encoding integer factorization problems where there is still room for fine tuning the parameters of B2C-consistency to achieve yet better performance.

The evaluation of the internal characteristics of our prototype preprocessing tool – namely the evaluation of the process of selection of the list of variables for consistency check – indicates a good match with theoretical expectations.

6. Conclusion and Future Work

In this paper, a new type of consistency called B2C-consistency (bounded $(2,k)$-consistency) for use in Propositional satisfiability (SAT) has been presented. This new consistency has been inspired by both global constraints and local consistency. Basically, it is $(2,k)$-consistency with the bounded number of search steps for proving inconsistency enriched by reasoning over complete sub-graphs of pair-wise conflicting literals. Reasoning over complete sub-graphs introduces a global aspect into proving inconsistency and it can improve the consistency enforcing process significantly especially in SAT instances encoding the well known P/H principle (pigeon/hole principle) and similar principles which are known to be difficult for a standard solving process based on search.

The whole design of new consistency is explained in the context of modeling SAT as a constraint satisfaction problem (CSP) using the so-called explicit literal encoding (that is, literal encoding with explicit clauses obtained by singleton unit propagation).

Next we investigated probabilistic properties of the so-called generalized P/H principle – particularly the expected number of satisfied (consistent) tuples of values with respect to a tuple of the selected variables for consistency check. The investigation showed that a certain distribution of the expected number of satisfied tuples is characteristic for the P/H principle where many inconsistent tuples

of values can be found. Therefore we proposed a process for the selection of variables which is trying to select variables so that the corresponding expected number of satisfied tuples of variables has a similar probabilistic distribution as in the case of the P/H principle. Using this process, we are trying to identify difficult sub-problems (such as the P/H principle) that can be yet resolved by B2C-consistency.

To evaluate our proposal we implemented B2C-consistency and the process of selection of variables within the prototype SAT preprocessing tool `preprocessSIGMA`. The experiments have confirmed that B2C-consistency and the variable selection process are beneficial and that we are able to select variables for consistency checks with similar probabilistic characteristics as in the case of the generalized P/H principle. The competitive evaluation on a set of 344 SAT instances from SATLib, SAT Competition 2003/2004 and 2007/2009 (mixture of satisfiable and unsatisfiable) showed that `preprocessSIGMA` delivers better results than the existent preprocessing tools `BCE`, `HyPre`, `LiVer`, and `Niver` which are based on local reasoning and comparable results to `saucy-3` based on symmetry breaking. In instances encoding the integer factorization problem `preprocessSIGMA` performed as far the best of all the tested preprocessing tools. Moreover, `preprocessSIGMA` has some advantages with respect to the comparable `saucy-3`. It is easier to implement – in `saucy-3`, graph isomorphism which, in itself, is a difficult problem needs to be solved – and it has many parameters that can be further fine tuned. Note that we have achieved a competitive performance despite the not well optimized implementation of `preprocessSIGMA`.

There are several interesting questions for future work. At present, we used a characterization of the distribution of an expected number of satisfied tuples of values with two parameters – the lower and the upper bound. It would be interesting to use more parameters to control the shape of the resulting distribution over all the consistency checks more precisely.

Another interesting investigation may be done with a repeated use of B2C-consistency. Consider a preprocessed instance to be preprocessed once again. Unfortunately, this approach is impractical at the current implementation stage as the setup of preprocessing is relatively time-consuming, and in order to preserve relatively acceptable competitiveness we cannot afford to run the process more than once. However, a more efficient implementation may change the situation.

References

1. **Aloul**, F. A., **Ramani**, A., **Markov**, I. L., **Sakallah**, K. A., *Solving Difficult SAT Instances in the Presence of Symmetry.* Proceedings of the 39th Design Automation Conference (DAC 2002), pp. 731-736, USA, (ACM Press, 2002, http://www.aloul.net/benchmarks.html, [March 2011]).
2. **Aloul**, F. **Markov**, I. L., **Sakallah**, K., Shatter: *Efficient Symmetry-Breaking for Propositional Satisfiability.* Proceedings of the Design Automation Conference (DAC 2003), pp. 836-839, (ACM Press, 2003, http://www.aloul.net/Tools/shatter/, [July 2011]).
3. **Aloul**, F. A., *SAT Benchmarks, Difficult Integer Factorization Problems.* Research web page, (http://www.aloul.net/benchmarks.html, [March 2011]).
4. **Anbulagan**, **Slaney**, J., *Multiple Preprocessing for Systematic SAT Solvers*. Proceedings of the 6th International Workshop on the Implementation of Logics, (CEUR-WS.org, 2006).
5. **Bacchus**, F., **Winter**, J., *Effective Preprocessing with Hyper-Resolution and Equality Reduction.* Proceedings of the Theory and Applications of Satisfiability Testing, 6th International Conference, (SAT 2003), pp. 341-355, LNCS 2919, (Springer 2004, http://www.cs.toronto.edu/~fbacchus/sat.html, [July 2011]).
6. **Bennaceur**, H., *The satisfiability problem regarded as constraint-satisfaction problem.* Proceedings of the 12th European Conference on Artificial Intelligence (ECAI 1996), pp. 155-159, (John Wiley and Sons, 1996).
7. **Bessière**, C., **Hebrard**, E., **Walsh**, T., *Local Consistencies in SAT.* Proceedings of the Theory and Applications of Satisfiability Testing, 6th International Conference (SAT 2003), pp. 299-314, LNCS 2919, (Springer, 2004).
8. **Biere**, A., **Heule**, M., **van Maaren**, H., **Walsh**, T., *Handbook of Satisfiability*. IOS Press, 2009.
9. **Bomze**, I. M., **Budinich**, M., **Pardalos**, P. M., **Pelillo**, M., *The maximum clique problem, Handbook of Combinatorial Optimization.* (Kluwer Academic Publishers, 1999).
10. **Cook**, S. A., *The Complexity of Theorem Proving Procedures.* Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971), pp. 151-158, ACM Press, 1971.
11. **Dechter**, R., *Constraint Processing.* (Morgan Kaufmann Publishers, 2003).
12. **Dowling**, W., **Gallier**, J., *Linear-time algorithms for testing the satisfiability of propositional Horn formulae.* Journal of Logic Programming, Volume 1 (3), pp. 267-284, (Elsevier Science Publishers, 1984).
13. **Eén**, N., **Sörensson**, N., *An Extensible SAT-solver.* Proceedings of Theory and Applications of Satisfiability Testing, 6th International Conference (SAT 2003), pp. 502-518, LNCS 2919, (Springer 2004, http://minisat.se/MiniSat.html, [July 2011]).

14. **Freuder**, E. C., *A Sufficient Condition for Backtrack-Free Search.* Journal of the ACM, Volume 29, pp. 24-32, (ACM Press, 1982).

15. **Golumbic**, M. C., *Algorithmic Graph Theory and Perfect Graphs.* (Academic Press, 1980).

16. **Heule**, M., **Järvisalo**, M., **Biere**, A., *Clause Elimination Procedures for CNF Formulas.* Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference (LPAR 2010), pp. 357-371, LNCS 6397, (Springer 2010).

17. **Hoos**, H. H., *Programming by optimization.* Communications of the ACM, Volume 55(2), pp. 70-80, (ACM Press, 2012).

18. **Hoos**, H. H., **Stützle**, T., *SATLib: An Online Resource for Research on SAT.* Proceedings of Theory and Applications of Satisfiability Testing, 4th International Conference (SAT 2000), pp.283-292, (IOS Press, 2000, http://www.satlib.org, [March 2011]).

19. **Jackson**, P., **Sheridan**, D., *Clause Form Conversions for Propositional Circuits.* Theory and Applications of Satisfiability Testing, 7th International Conference (SAT 2004), Revised Selected Papers, pp. 183–198, Lecture Notes in Computer Science 3542, Springer 2005.

20. **Järvisalo**, M., **Biere**, A., **Heule**, M., *Blocked Clause Elimination.* Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference (TACAS 2010), pp. 129-144, LNCS 6015, Springer, 2010.

21. **Katebi**, H., **Sakallah**, K. A., **Markov**, I., L., *Symmetry and Satisfiability: An Update.* Proceedings of Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference (SAT 2010), pp. 113-127, LNCS 6175, Springer 2010.

22. **Papadimitriou**, C., *Computational Complexity.* (Addison Wesley, 1994).

23. **Petke**, J., **Jeavons**, P., *Local Consistency and SAT-Solvers.* Proceedings of the Principles and Practice of Constraint Programming - 16th International Conference (CP 2010), pp. 398-413, Lecture Notes in Computer Science 6308, (Springer 2010).

24. **Seidel**, R., *On the Complexity of Achieving K-Consistency.* Technical Report, University of British Columbia Vancouver, BC, (Canada,1983).

25. **Subbarayan**, S., **Pradhan**, D., K., *NiVER: Non Increasing Variable Elimination Resolution for Preprocessing SAT Instances*. Proceedings of The 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), pp. 276-291, LNCS 3542, (Springer 2005, http://www.itu.dk/people /sathi/niver.html, [July 2011]).

26. **Surynek**, P., *Solving Difficult SAT Instances Using Greedy Clique Decomposition.* Proceedings of the 7th Symposium on Abstraction, Reformulation, and Approximation (SARA 2007), LNAI 4612, pp. 359-374, (Springer, 2007).

27. **Surynek**, P., *sigmaSAT / preprocessSIGMA - a collection of experimental SAT processing tools.* Research web page, Charles University in Prague, 2011, http://ktiml.mff.cuni.cz/~surynek/index.html.php?select=software&product=sigmasat [July 2011].

28. **Torán**, J., *The Hardness of Graph Isomorphism.* SIAM Journal on Computing, , pp. 1093-1108, volume 33, number 5, SIAM, 2004.

29. **Walsh**, T., *SAT vs. CSP*. Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming, 441-456, LNCS 1894, Springer Verlag, 2000.

Articles in proceedings of leading conferences

Pavel **Surynek**: *Reduced Time-Expansion Graphs for Solving Cooperative Path Finding Sub-optimally.* Proceedings of the 24rd International Joint Conference on Artificial Intelligence (IJCAI 2015), Buenos Aires, Argentina, IJCAI/AAAI Press, 2015.

Reduced Time-Expansion Graphs and Goal Decomposition for Solving Cooperative Path Finding Sub-optimally

Pavel Surynek

Charles University Prague
Faculty of Mathematics and Physics
Department of Theoretical Computer Science and Mathematical Logic
Malostranské náměstí 25, Praha, 118 00, Czech Republic
pavel.surynek@mff.cuni.cz

Abstract. Solving cooperative path finding (CPF) by translating it to propositional satisfiability represents a viable option in highly constrained situations. The task in CPF is to relocate agents from their initial positions to given goals in a collision free manner. In this paper, we propose a reduced time expansion that is focused on makespan sub-optimal solving. The suggested reduced time expansion is especially beneficial in conjunction with a goal decomposition where agents are relocated one by one.

***Keywords*:** cooperative path finding, propositional satisfiability, time expansion graphs, vertex disjoint paths

1. Introduction and Motivation

The problem of cooperative path-finding (CPF) [Kornhauser *et al.*, 1984; Silver, 2005, Ryan, 2008] is a graph theoretical abstraction for many real life problems where the task is to cooperatively relocate a group of robots or other movable objects in a collision free manner. Each agent of the group is given its initial and goal position in the environment. The problem consists in constructing a spatial temporal plan for each agent by which it can relocate from its initial position to the given goal. The environment where agents move is modeled as an undirected

graph [Kornhauser *et al.*, 1984] where vertices represent locations and edges represent possibility of relocation between two locations.

Agents are represented as abstract items placed in vertices while at most one agent is located in each vertex. An agent can instantaneously relocate itself to the neighboring vertex assumed the target vertex is unoccupied and no other agent is trying to enter the same target vertex.

In this research, we further develop solving of CPF by translating it to propositional satisfiability (SAT) [Biere *et al.*, 2009]. Recent propositional encodings [Surynek, 2012a, 2012b, 2013, 2014] of CPF are based on time expansion of the graph modeling the environment so that the encoding is able to represent arrangements of agents over the graph at all the time steps up to the final one. Since there may be many time-steps before all the agents reach their goals, these encodings may become extremely large and hence unsolvable in reasonable time. We are trying to overcome this limitation by reducing the expansion of the graph in this work.

1.1. *Context of Related Works*

The approach to solve CPF by reducing it to SAT has multiple alternatives. There exist algorithms based on search that find makespan optimal or near optimal solutions. The seminal work in this category is represented by Silver's WHCA* algorithm [Silver, 2005]. Recent contributions include OD+ID [Standley and Korf, 2011], which is a combination of A* and powerful agent independence detection heuristics, and ICTS [Sharon et al., 2013] which employs the concept of increasing cost tree (instead of makespan, the total cost of solution is optimized). Other approaches resolve conflicts among robot trajectories when avoidance is necessary [Čáp *et al.*, 2013; Barer et al., 2014; Wagner and Choset, 2015].

Fast polynomial time algorithms for generating makespan suboptimal solutions include PUSH-AND-ROTATE [de Wilde *et al.*, 2014]. The drawback of these algorithms is that their solutions are dramatically far from the optimum.

Translation of CPF to a different formalism, namely to answer set programming (ASP), has been suggested in [Erdem *et al.*, 2013]. Integer programming (IP) as the target formalism has been also used [Yu and LaValle, 2013]. The choice of SAT as the target formalism is very common in domain independent planning where the idea of time expansion [Kautz and Selman, 1999; Huang *et al.*, 2010] and its reductions [Wehrle and Rintanen, 2007] are studied.

2. Formal Definition of CPF

An arbitrary **undirected graph** $G = (V, E)$ can be used to model the environment where agents are moving. The placement of agents in the environment is modeled by assigning them vertices of the graph. Let $A = \{a_1, a_2, \ldots, a_\mu\}$ be a finite set of *agents*, then, an arrangement of agents in vertices of graph G is fully described by a *location* function $\alpha: A \longrightarrow V$. At most **one agent** can be located in each vertex; that is α is uniquely invertible.

Definition 1 (COOPERATIVE PATH FINDING). An instance of *cooperative path-finding* problem is a quadruple $\Sigma = [G = (V, E), A, \alpha_0, \alpha_+]$ where location functions α_0 and α_+ define the initial and the goal arrangement of a set of agents A in G respectively. □

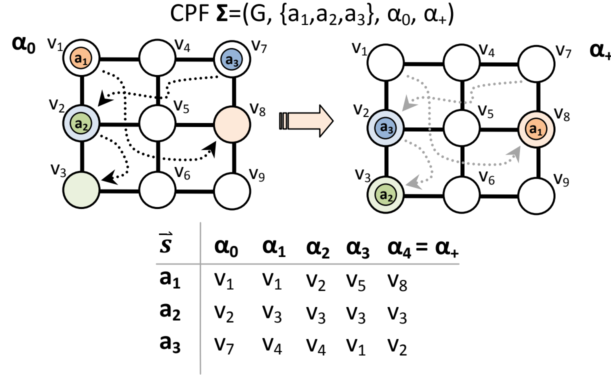
The dynamicity of the model supposes a discrete time divided into time steps. An arrangement α_i at the i-th time step can be transformed by a transition action which instantaneously moves agents in the non-colliding way to form a new arrangement α_{i+1}. The transition between α_i and α_{i+1} must satisfy the following *validity conditions*:

- $\forall a \in A$ either $\alpha_i(a) = \alpha_{i+1}(a)$ or $\{\alpha_i(a), \alpha_{i+1}(a)\} \in E$ (1)
  (agents move along edges or not move at all),
- $\forall a \in A \;\; \alpha_i(a) \neq \alpha_{i+1}(a) \Rightarrow (\forall b \in A \; \alpha_i(b) \neq \alpha_{i+1}(a))$ (2)
  (agents move to vacant vertices only), and
- $\forall a, b \in A \;\; a \neq b \Rightarrow \alpha_{i+1}(a) \neq \alpha_{i+1}(b)$ (3)
  (no two agents enter the same target/unique invertibility of resulting arrangement).

The task in cooperative path finding is to transform α_0 using above valid transitions to α_+. An illustration of CPF and its solution is depicted in Figure 1.

Definition 2 (SOLUTION, MAKESPAN). A *solution* of a *makespan* m to a cooperative path finding instance $\Sigma = [G, A, \alpha_0, \alpha_+]$ is a sequence of arrangements $\vec{s} = [\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_m]$ where $\alpha_m = \alpha_+$ and α_{i+1} is a result of valid transition from α_i for every $i = 1,2, \ldots, m - 1$. □

It is known that finding makespan optimal solution to CPF is NP-hard [Ratner and Warmuth, 1986].



$\vec{s}$	α_0	α_1	α_2	α_3	$\alpha_4 = \alpha_+$
a_1	v_1	v_1	v_2	v_5	v_8
a_2	v_2	v_3	v_3	v_3	v_3
a_3	v_7	v_4	v_4	v_1	v_2

Figure 1. ***Cooperative path-finding (CPF)** on a 4-connected grid.* The task is to relocate three agents a_1, a_2, and a_3 to their goal vertices so that they do not collide with each other. A solution $\vec{s}$ of makespan 4 is shown.

3. (Sub)optimization in CPF via SAT

The approach we are suggesting here to obtain parameter optimal solutions is to employ *propositional satisfiability* (SAT) solving as the key technology. This approach has been already successfully applied in obtaining makespan optimal plans in domain-independent planning [Kautz and Selman, 1999; Huang *et al.*, 2010] as well as in CPF [Surynek, 2013].

Algorithm 1. *SAT-based parameter optimal CPF solving – sequential increasing strategy.* The algorithm sequentially finds the smallest possible makespan η for that a propositional encoding of a given CPF $\Sigma = (G, A, \alpha_0, \alpha_+)$ is solvable.

input: Σ – a CPF instance
output: a pair consisting of the optimal parameter and corresponding parameter optimal solution

function *Find-Optimal-Parameter* $(\Sigma = (\mathrm{G}, A, \alpha_0, \alpha_+))$: **pair**
1: $\eta \leftarrow 1$
2 **loop**
3: $F(\Sigma, \eta) \leftarrow$ *Encode-CPF-as-SAT* (Σ, η)
4: **if** *Solve-SAT* $(F(\Sigma, \eta))$ **then**
5: **let** f be a satifying valuation of $F(\Sigma, \eta)$
6: **return** (η, f)
7: $\eta \leftarrow \eta + 1$
8: **return** $(\infty, \emptyset)$

In case of CPF, a propositional formula $F(\Sigma, \eta)$ such that it is satisfiable if and only if a given CPF Σ with makespan bound η is solvable can be constructed. Being able to construct such a formula $F(\Sigma, \eta)$ one can obtain the optimal makespan for the given CPF Σ by asking multiple queries whether formula $F(\Sigma, \eta)$ is satisfiable with different makespan bounds η.

Various strategies of the parameter for queries exist for getting the parameter optimal solution. The simplest is to try sequentially makespans $\eta = 1,2, \ldots$ until η is equal to the optimum (minimum). This strategy will be further referred as *sequential increasing*. Pseudo-code of the strategy is listed as Algorithm 1.

4. Reduced Time Expansion Graph

The main drawback of makespan optimal CPF solving via SAT is the large size of the formulae that encode the optimization questions [Surynek, 2013, 2014]. The size of encoding formulae becomes especially prohibitive when they encode questions if a solution with a large makespan exists. This is due to the fact that existing encodings expands the graph modeling the environment over the time up to the given makespan bound η. At each time step of the expansion arrangement of agents over the graph is represented and constraints ensure that only transitions conforming to validity conditions are possible between arrangements at consecutive time steps.

Our idea hence was to **reduce** the time expansion with possible **relaxation** of the requirement of makespan optimality of the solution. The key observation is that if there is no need of any complex avoidance between agents (there is no need to visit a single vertex multiple times), no time expansion of the graph is necessary at all. The question if there is a solution (not necessarily makespan optimal) can be stated as a question of existence of vertex disjoint paths connecting initial positions of agents with their goals in the original graph. Translating of this question into SAT is possible as well.

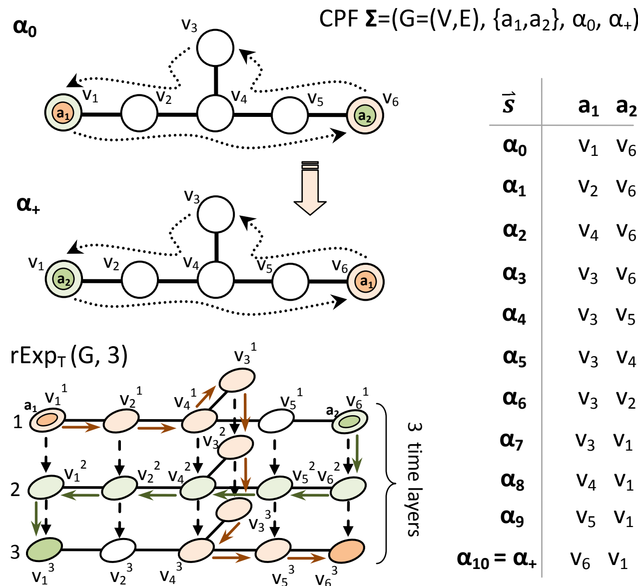
Nevertheless, in real situations movement interactions among agents require complex avoidance. A single vertex may need to be visited multiple times. This led us to the suggestion of a concept of *reduced time expansion graph*, which combines the expansion reduction with ability to represent complex avoidance.

Definition 3 (REDUCED TIME EXPANSION GRAPH - $\text{rExp}_{\text{T}}(G, \vartheta)$). Let $G = (V, E)$ be an undirected graph and $\vartheta \in \mathbb{N}$. A *reduced time expansion graph* with ϑ *time layers* associated with G is a directed graph $\text{rExp}_{\text{T}}(G, \vartheta) = (V \times$

$\{1,2,\ldots,\vartheta\}, E')$ where $E' = \{([u,l],[v,l]) | \{u,v\} \in E; l = 1,2,\ldots,\vartheta\} \cup \{([v,l],[v,l+1]) \mid l = 1,2,\ldots,\vartheta - 1\}$. □

Note, that for each original undirected edge there are two directed arcs in both directions in the reduced time expansion graph. A *time-layer* in the reduced time expansion graph is an induced sub-graph of $\text{rExp}_\text{T}(G,\vartheta)$ over the set of vertices $V \times \{l\}$ for a given $l \in \{1,2,\ldots,\vartheta\}$.

Solving of CPF $\Sigma = [G, A, \alpha_0, \alpha_+]$ can be viewed as a search for *vertex disjoint paths* in $\text{rExp}_\text{T}(G,\vartheta)$ that connect initial positions and goals in the first and the last time-layer respectively provided that the number of time-layers ϑ is sufficiently high. The idea is illustrated in Figure 2.



$\vec{s}$	a_1	a_2
α_0	v_1	v_6
α_1	v_2	v_6
α_2	v_4	v_6
α_3	v_3	v_6
α_4	v_3	v_5
α_5	v_3	v_4
α_6	v_3	v_2
α_7	v_3	v_1
α_8	v_4	v_1
α_9	v_5	v_1
$\alpha_{10} = \alpha_+$	v_6	v_1

Figure 2. *An example of CPF and its solving through* ***reduced time expansion graph***. A reduced time expansion graph $\text{rExp}_\text{T}(G,3)$ consisting of 3 time layers is build for a given CPF Σ. A solution to Σ corresponds to a collection of vertex disjoint paths connecting the initial positions agents in the first layer with their goal positions in the last time layer.

4.1. ϑ-RELAXED *Propositional Encoding*

The correspondence between the existence of vertex disjoint paths and the existence of a solution of CPF established in the previous section provides a guide how to design required propositional encoding. We merely need to design a prop-

ositional formula preferably in *conjunctive normal form* (CNF) [Biere *et al.*, 2009] that is satisfiable if and only if vertex disjoint paths connecting initial position and goals exist in $\text{rExp}_\text{T}(G, \vartheta)$ for $\vartheta \in \mathbb{N}$.

Intuitivelly, the size and the structure of the resulting formula matters when it is solved by a SAT solver. Our choice was to design an encoding that is space efficient and contains short clauses. Note that short clauses support *unit propagation* [Biere *et al.*, 2009].

The encoding is separated into two parts. The first part is purely propositional and consists of variables that express selection of vertices and edges into paths – this can be also regarded as occupancy/selection of path by a flow of commodity. The inspiration for this design comes from the theory of network flows [Ahuja *et al.*, 1993]. The absence of necessity to distinguish between individual agents enables expressing the requirement that paths should be vertex disjoint as simple capacity constraints.

The distinguishable agents are treated in the second part of the model where a bit vector using binary encoding is associated with each vertex in $\text{rExp}_\text{T}(G, \vartheta)$ to express what agent is occupying that. The benefit of using bit-vectors is that equality can be easily expressed over them. Both parts are put together by introducing a constraint that requires occupation by the same agent at both ends of a selected edges. Formally, the encoding – which we called ϑ-RELAXED – is introduced in the following definition.

Definition 4 (ϑ-RELAXED encoding - $F_{\vartheta-RE}(\Sigma)$). Let $\Sigma = [G, A, \alpha_0, \alpha_+]$ be a CPF with $G = (V, E)$. A *ϑ-RELAXED encoding* for CPF Σ consists of the following collections of variables for every time layer $l \in \{1,2, \ldots, \vartheta\}$: finite domain variables $\mathcal{A}_v^l \in \{0,1, \ldots, \mu\}$ for every $v \in V$ (that are encoded as bit vectors), propositional variables $\mathcal{X}_v^l$ for every $v \in V$, and propositional variables $\mathcal{E}_{u,v}^l$ for every ordered pair u, v such that $\{u, v\} \in E$ (that is, for a single edge $\{u, v\} \in E$ and l we have two propositional variables $\mathcal{E}_{u,v}^l$ and $\mathcal{E}_{v,u}^l$). Additionally, there is a set of propositional variables $\mathcal{E}_v^l$ for every every $v \in V$ and $l \in \{1,2, \ldots, \vartheta - 1\}$ representing interconnections between time layers. Constraints of ϑ-RELAXED encoding are as follows:

- $\mathcal{A}_v^l \neq 0 \Rightarrow \mathcal{X}_v^l$ for every $v \in V$ and $l \in \{1,2, \ldots, \vartheta\}$ (4)
  (if there is some agent in a vertex then the vertex is non-empty)

- $\mathcal{E}_{u,v}^{l} \Rightarrow \mathcal{X}_{u}^{l} \wedge \mathcal{X}_{v}^{l}$ for every $\{u,v\} \in E$ and $l \in \{1,2,\dots,\vartheta\}$ (5)

  $\mathcal{E}_{v}^{l} \Rightarrow \mathcal{X}_{v}^{l} \wedge \mathcal{X}_{v}^{l+1}$ for every $v \in V$ and $l \in \{1,2,\dots,\vartheta-1\}$ (6)

  (<u>if an edge within a time layer or between time layers is non-empty then its both ends are non-empty</u>)
- $\bigwedge_{u|\{u,v\}\in E} \neg\mathcal{E}_{u,v}^{1}$ for every $v \in V$ such that $(\exists a \in A)\alpha_0(a) = v$ (7)

  (<u>for every source vertex at the first time layer all the incoming directed edges are empty</u>)

  $\bigwedge_{v|\{u,v\}\in E} \neg\mathcal{E}_{u,v}^{\vartheta}$ for every $u \in V$ such that $(\exists a \in A)\alpha_+(a) = u$ (8)

  (<u>for every destination vertex at the last time layer all the outgoing directed edges are empty</u>)
- $\mathcal{E}_{u,v}^{l} \Rightarrow \mathcal{A}_{u}^{l} = \mathcal{A}_{v}^{l}$ for every $\{u,v\} \in E$ and $l \in \{1,2,\dots,\vartheta\}$ (9)

  $\mathcal{E}_{v}^{l} \Rightarrow \mathcal{A}_{v}^{l} = \mathcal{A}_{v}^{l+1}$ for every $v \in V$ and $l \in \{1,2,\dots,\vartheta-1\}$ (10)

  (<u>if an edge is non-empty then there is the same agent at its both endpoints</u>)
- $\mathcal{X}_{u}^{l} \Rightarrow \bigvee_{v|\{u,v\}\in E} \mathcal{E}_{u,v}^{l} \vee \mathcal{E}_{u}^{l}$ for every $u \in V$ and $l \in \{1,2,\dots,\vartheta-1\}$ (11)

  $\sum_{v|\{u,v\}\in E} \mathcal{E}_{u,v}^{l} + \mathcal{E}_{u}^{l} \leq 1$ (12)

  (<u>if a vertex is non-empty at a time layer other than the last one then exactly one of its outgoing edges is non-empty as well</u>)

  $\mathcal{X}_{u}^{\vartheta} \Rightarrow \bigvee_{v|\{u,v\}\in E} \mathcal{E}_{u,v}^{\vartheta}$ for every $u \in V$ such that (13)

  $\sum_{v|\{u,v\}\in E} \mathcal{E}_{u,v}^{\vartheta} \leq 1$ $(\forall a \in A)\alpha_+(a) \neq u$ (14)

  (<u>if a non-destination vertex at the last time layer is non-empty then exactly one of its outgoing edges is non-empty as well</u>)
- $\mathcal{X}_{v}^{l} \Rightarrow \bigvee_{u|\{u,v\}\in E} \mathcal{E}_{u,v}^{l} \vee \mathcal{E}_{v}^{l-1}$ for every $v \in V$ and $l \in \{2,3,\dots,\vartheta\}$ (15)

  $\sum_{u|\{u,v\}\in E} \mathcal{E}_{u,v}^{l} + \mathcal{E}_{v}^{l-1} \leq 1$ (16)

(if a vertex is non-empty at a time layer other than the first one then exactly one of its incoming edges is non-empty as well).

$\mathcal{X}_v^1 \Rightarrow \bigvee_{u|\{u,v\}\in E} \mathcal{E}_{u,v}^1$ for every $v \in V$ such that (17)

$\sum_{u|\{u,v\}\in E} \mathcal{E}_{u,v}^1 \leq 1$ $(\forall a \in A)\alpha_0(a) \neq v$ (18)

(if a non-source vertex at the first layer is non-empty then exactly one of its incoming edges is non-empty as well). □

Initial and goal arrangements are expressed as constraints over variables of the first and the last time layer. Note that some agents do not need to be assigned any goal if we do not care about their final positions.

The resulting formula of the *ϑ-RELAXED* encoding in the CNF form will be denoted as $F_{\vartheta-RE}(\Sigma)$. Without proof let us summarize the size of the encoding.

Proposition 1 ***(ϑ-RELAXED ENCODING SIZE).*** *The number of propositional variables in* $F_{\vartheta-RE}(\Sigma)$ *is* $\mathcal{O}(\vartheta \cdot (|V| \cdot \lceil \log_2(\mu+1) \rceil + |E|))$ *and the number of clauses is* $\mathcal{O}(\vartheta \cdot ((|V| + |E|) \cdot \lceil \log_2(\mu+1) \rceil + |V|^3))$. ■

A set $\Pi = \{\pi_1, \pi_2, \dots, \pi_\mu\}$ of vertex disjoint paths in $\mathrm{rExp_T}(\mathrm{G}, \vartheta)$ so that π_i connects $[\alpha_0(a_i),1]$ with $[\alpha_+(a_i), \vartheta]$ for $i = 1,2,\dots,\mu$ exists if and only if $F_{\vartheta-RE}(\Sigma)$ is satisfiable. The extraction of a solution of CPF Σ from a satisfying valuation of $F_{\vartheta-RE}(\Sigma)$ is shown using pseudo-code as Algorithm 2.

The algorithm tracks moves of agents towards their exits from the current time layer of the reduced time expansion graph during which the solution α is recorded. Note, that in each time layer the time step at which agents exit the layer is synchronized among all the agents (that is, agents exit at the same time step). It may therefore occur that agents wait for the last agent to finish its movements in the layer before they exit the layer together into the next one. The algorithm allows us to state the following theorem (proof is omitted).

Theorem 1 ***(SOLUTION OF*** Σ *AND* $F_{\vartheta-RE}(\Sigma)$ *SATISFACTION).* *A solution of a CPF* $\Sigma = (G, A, \alpha_0, \alpha_+)$ *with* $A = \{a_1, a_2, \dots, a_\mu\}$ *exists if and only if there ist* $\vartheta \in \mathbb{N}$ *for that formula* $F_{\vartheta-RE}(\Sigma)$ *is satisfiable.* ■

The original goal to reduce the size of the encoding by reducing the expansion of G is fulfilled by the fact that ϑ-RELAXED encoding needs no more time-layers than encodings for makespan optimal CPF solving. Moreover, there are cases where ϑ-RELAXED encoding needs significantly fewer time expansions – see

example in Figure 2 where 3 time expansions are needed in ϑ-RELAXED encoding while makespan optimal encodings need 8 time expansions.

Algorithm 2. *Solution extraction algorithm for ϑ-RELAXED encoding.* A sequence of arrangements of agents forming a solution of given CPF Σ is extracted from satisfying valuation f of formula $F_{\vartheta-RE}(\Sigma)$ representing ϑ-RELAXED encoding of Σ.

input: Σ – an instance of CPF
ϑ – the number of time layers in ϑ-RELAXED encoding
f – a satisfying valuation of $F_{\vartheta-RE}(\Sigma)$
output: makespan and sequence of arrangements of agents forming the solution $\alpha_0, \alpha_1, \ldots, \alpha_+$

```
function Extract-Solution-ϑ-RELAXED (Σ = [G = (V,E), A, α_0, α_+], ϑ, f): pair
1:   η_max ← 0   // time step at which movements at a time layer
                 // are finished
2:   for each l = 1,2,...,ϑ do
3:   |   η_min ← η_max       // time step at which movements
     |                       // at a time layer start
4:   |   for each a ∈ A do
5:   |   |   η ← η_min
6:   |   |   u ← α_{η_min}(a)
7:   |   |   while (l ≠ ϑ and f(ℰ_u^l) = FALSE)
     |   |   |   or (l = ϑ and u ≠ α_+(a)) do
8:   |   |   |   α_{≥η}(a) ← u   // agent a will be located in u
     |   |   |                   // at all the time steps ≥ η
9:   |   |   |   for each v ∈ V such that {u,v} ∈ E do
10:  |   |   |   |   if f(ℰ_{u,v}^l) = TRUE then
11:  |   |   |   |   |   u ← v
12:  |   |   |   η ← η + 1
13:  |   |   η_max ← max(η_max, η)
14:  |   |   α_{≥η}(a) ← u
15:  return (η_max, [α_0, α_1, ..., α_{η_max}])
```

Proposition 2 ***(ADVANTAGE OF ϑ-RELAXED ENCODING).*** *Let η be an optimal makespan achievable in a CPF Σ. Then $F_{\vartheta-RE}(\Sigma)$ is solvable for $\vartheta \leq \eta$. Moreover, there exists a CPF instance Σ where strict inequality $\vartheta < \eta$ holds.* ■

The number of time layers in ϑ-RELAXED encoding that grants finding a solution corresponds rather to the intensity of interactions among agents. Hence to further reduce the size of the encoding via reducing the number of time layers we suggest decomposing solving of a given CPF Σ into solving multiple CPFs in which intesity of interactions among agents is low and thus they can be solved by satifying ϑ-RELAXED encoding formulae consisting of few time layers.

The suggested decomposition corresponds to placing agents to their goals one by one while individual CPFs represents relocating a single agent where positions of previously placed agents are preserved. The process is called UniAGENT solving and it is formally described as Algorithm 3.

Without proof let us state that the UniAGENT method is **sound**; that is, it always finds a solution provided a solution exists. This is due to the fact, that we do constrain only agents that have been placed so far while remaining agents can be placed arbitrarily. This in theory tells that all the sub-goals determined by single agent placement are feasible.

Algorithm 3. *UniAGENT SAT-based CPF solving.* Agents (robots) are placed to their goals one by one. Relocation of a single agent to its goal is solved as an individual CPF using ϑ-RELAXED encoding where already placed agents preserve their positions. Relatively small difference between the initial arrangement and goal in single agent relocation CPFs allows to solve them with few time layers in the reduced time expansion graph.

input: Σ – an instance of CPF
output: makespan and a sequence of arrangements of agents
of arrangements of agents forming the solution

---

function *Solve-UniAGENT* ($\Sigma = [G = (V, E), A, \alpha_0, \alpha_+]$): **pair**
1: **let** $A = \{a_1, a_2, \dots, a_\mu\}$
2: $\eta_{\max} \leftarrow 0$
3: **for each** $i = 1,2, \dots, \mu$ **do**
4: $\beta_0 \leftarrow \alpha_{\eta_{\max}}$
5: **for each** $j = 1,2, \dots, i-1$ **do**
6: $\beta_+(a_j) \leftarrow \alpha_{\eta_{\max}}(a_j)$
7: $\beta_+(a_i) \leftarrow \alpha_+(a_i)$
8: $(\vartheta, f) \leftarrow$ *Find-Optimal-Parameter* ($\Phi = [G, A, \beta_0, \beta_+]$)
9: $(\eta, \vec{s}) \leftarrow$ *Extract-Solution-ϑ-RELAXED* (Φ, ϑ, f)
10: **for each** $k = 0,1, \dots, \eta - 1$ **do**
11: $\alpha_{\eta_{\max}+k} \leftarrow \vec{s}[k]$
12: $\eta_{\max} \leftarrow \eta_{\max} + \eta$
13: **return** ($\eta_{\max}$, $[\alpha_0, \alpha_1, \dots, \alpha_{\eta_{\max}}]$)

In our minor experiments, we found that the ϑ-RELAXED encoding is very easy to solve if there are few time layers but it gets rapidly harder with the increasing number of time layers. The number of time layers necessary to reach the solvability when a single agent is relocated is typically very low (usually 1 to 3 time layers). Moreover, the makespan of solutions generated by the UniAGENT solving process is similar to that of generated by solving the ϑ-RELAXED encoding where all the agents are relocated at once in cases where we managed to solve

the ϑ-RELAXED encoding. These observations together justifies the use of the new encoding as suggested in the UniAGENT solving process.

5. Experimental Evaluation

Series of experiments have been conducted in order to evaluate the suggested propositional ϑ-RELAXED encoding and UniAGENT solving process that is based on it.

The comparison has been done with existent encodings for makespan optimal CPF solving – INVERSE, ALL-DIFFERENT, DIRECT, MATCHING, and SIMPLIFIED [Surynek, 2012a, 2012b, 2014]. To include other than SAT-based methods, the comparison with A*-based OD+ID [Standley and Korf, 2011] for makespan optimal solving is also presented. Makespan suboptimal methods are represented by WHCA* [Silver, 2005] in our comparison.

We used benchmarks suggested in [Silver, 2005] which consist of randomly generated CPF instances over **4-connected grids** with randomly placed obstacles. There are also randomly placed **obstacles** by which 20% of all the vertices are occupied. An important module in the whole solving process is a SAT solver. `Glucose` version `3.0` [Audemard and Simon, 2013] has been used in the experimental evaluation.

5.1. *Encoding Size Comparison*

The important characteristic of propositional formulae with respect to the speed of their solving is their size while small is preferable (the size is represented by the *number of variables and clauses* in our case).

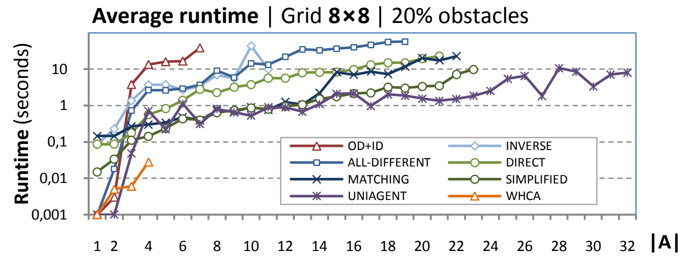
Selected results are shown in Table 1. Size measurement is done on 4-connected grid and for various numbers of agents in the environment. For each number of agents **10 random instances** were generated and average value for each characteristic is presented.

It can be observed from presented results that the ϑ-RELAXED encoding is the smallest in terms of the number of clauses and the second smallest in terms of the number of variables just after the MATCHING encoding. The average clause length also indicates that most of clauses are binary.

Note, that formulae for all the encodings were generated with the same number of time-layers. In most cases however, ϑ-RELAXED encoding needs fewer time-layers to achieve solvability.

Table 1. ***Size comparison*** *of encodings over 8×8 grid.* INVERSE, ALL-DIFFERENT, DIRECT, MATCHING, SIMPLIFIED [Surynek, 2012a, 2012b, 2014] and ϑ-RELAXED encodings are compared. CPF instances are generated over the 4-connected grid of size 8×8 with 20% of cells occupied by obstacles. Makespan bound η and the number of time layers in reduced time expansion graph ϑ is always 16. The number of *variables* and *clauses*, the *ratio* of the number of clauses and the number of variables, and the *average clause length* are listed for different sizes of the of agents A. The advantage of ϑ-RELAXED encoding is that it is relatively small compared to other encodings.

Grid 8×8 **|Agents|**		INVERSE		ALL-DIFFERENT		DIRECT		MATCHING		SIMPLIFIED		θ-RELAXED	
1	#Variables #Clauses *Ratio* *Length*	8 358.7 31 327.9	*3.748* *2.616*	1 489.3 7 930.4	*5.325* *3.057*	**814.4** 23 241.9	*28.539* ***2.149***	4 520.3 25 881.1	5.710 2.441	1 628.8 **3 384.6**	*2.078* *2.550*	4 645.1 20 246.6	*4.358* *2.515*
4		10 019.5 55 437.0	*5.532* *2.641*	7 834.5 34 781.9	*4.440* *3.103*	**3 257.6** 115 934.3	*35.589* ***2.272***	6 181.1 43 171.0	6.984 2.640	4 072.0 **17 997.8**	*4.420* *2.374*	6 273.9 33 904.1	*5.404* *2.660*
16		11 680.3 91 344.5	*7.820* *3.127*	67 088.3 216 745.4	*3.231* *3.147*	13 030.4 840 540.6	*64.506* *2.505*	**7 841.9** 72 259.3	9.215 3.315	13 844.8 150 259.2	*10.853* ***2.180***	7 902.7 **47 324.6**	*5.988* *2.714*
32		12 510.7 122 170.3	*9.765* *3.733*	230 753.0 646 616.2	*2.802* *3.168*	26 060.8 2 738 584.7	*105.084* *2.621*	**8 672.3** 99 675.5	11.494 4.045	26 875.2 510 672.1	*19.002* ***2.111***	8 717.1 **53 697.0**	*6.159* *2.722*

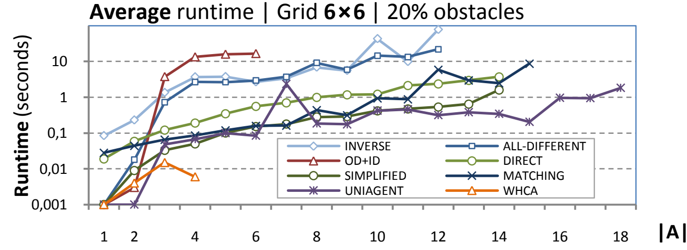


|A|	1	4	8	12	16	20	24	28	32
η	5.3	8.4	11.0	11.7	12.4	12.3	-	-	-
ω	5.6	9.3	-	-	-	-	-	-	-
ϑ	9.3	15.8	33.0	49.3	83.4	96.1	131.4	154.1	201.7

Figure 3. *Runtime and makespan comparison over 8×8 grid.* UniAGENT and WHCA* produce makespan sub-optimal solutions; all other methods are makespan optimal. Evaluation of runtime and makespan was done for the growing number of agents (timeout is 256 seconds). Average optimal makespan is shown as η; ϑ and ω are average makespans of UniAGENT and WHCA* respectively.

5.2. *Runtime Evaluation*

Runtime tests were done over 4-connected grids with growing number of agents. The *timeout* has been set to **256 seconds** and for each number of agents **10 random instances** were solved while runtime was recorded – average runtime is presented.



$\|A\|$	1	2	4	6	8	12	14	16	18
η	4.2	4.9	5.6	7.0	7.4	7.9	8.6	-	-
ω	4.3	5.3	5.7	-	-	-	-	-	-
ϑ	5.7	8.5	11.1	16.7	30.2	43.1	49.3	50.5	87.3

Figure 4. *Runtime and makespan comparison over 6×6 grid.* The UniAGENT solving is almost by order of magnitude faster than second best method for higher number of agents.

Runtime results are presented in
Figure **3** and
Figure **4**. Average optimal makespan and average sub-optimal makespan obtained with the UniAGENT and WHCA* methods are also shown. It can be observed that OD+ID and WHCA* although performing as best for small number of agents, quickly reaches the timeout as the number of agents grows. UniAGENT method scales up as the best for growing number of agents though the makespan is up to several times longer than the optimum. Up to 30 agents (occupancy 83%) and up to 48 agents (occupancy of 75%) can be solved in 6×6 and 8×8 grid respectively with no obstacles within the timeout of 1.0 minute.

Table 2. *Makespan comparison with **domain independent planners**.* Suboptimal planners LPG-td and SGPLAN managed to solve instances over the 6×6 grid with 20% obstacles with up to 6 agents within the timeout of 256 seconds. UNIAGENT solver generates solutions of shorter makespan and is much faster.

$\|A\|$ in 6×6	1	2	3	4	5	6	7
LPG-td	17.2	7.6	18.5	16.2	22.7	134.1	-
SGPLAN	7.2	9.8	16.7	15.1	23.4	-	-
UNIAGENT	5.7	8.5	12.3	11.1	15.9	16.7	20.5

Motivated by experiments presented in [Standley and Korf, 2011], we also tried to solve (N^2-2)-puzzles by the UniAGENT solver; that is, 4-connected grids with two blanks (two blanks grant that instances are solvable). In these situations, A*-based solvers relying on independence detection such as OD+ID and

MGS1 do not scale well. The $(3^2 - 2)$-puzzles were solved in less than 1.0 second by UniAGENT solver. The $(4^2 - 2)$-puzzles needed approximately 10 seconds. Larger puzzles have not been solved under 1.0 minute.

We also made comparison with several domain independent planners including SAT-based makespan optimal SATPLAN [Kautz and Selman, 1999] and SASE [Huang *et al.*, 2010] and makespan suboptimal LPG-td [Gerevini *et al.*, 2008] and SGPLAN [Hsu *et al.*, 2006]. Planners were run on instances over 6×6 grid with 20% obstacles containing few agents - part of results is shown in Table 2. SATPLAN and SASE performed orders of magnitude worse than SAT-based solving with referred domain dependent encodings (thus not presented). Makespan suboptimal planners LPG-td and SGPLAN performed much better but still do not scale up. Moreover, they tend to generate worse makespans than the UniAGENT method.

6. Conclusions

The concept of *reduced time expansion* graph and ϑ-RELAXED propositional encoding of CPF based on this graph have been introduced. The search for a solution of CPF is reduced to the search of **vertex disjoint paths** in reduced time expansion graph which is done via SAT solving. In order to maximally reduce the size of the propositional encoding, the search for a goal arrangement is decomposed into multiple searches for sub-goals which correspond to placement of a single agent.

Experimental evaluation indicates that the novel CPF solving method - called UniAGENT solver - **scales up** better for higher number of agents than comparable makespan suboptimal search-based method WHCA*. The relaxation from the requirement on the makespan optimality allowed significant **runtime improvement** compared to other propositional encodings and related SAT-based solving schemes. This advanced applicability of SAT-based CPF solving in **highly constrained** situations towards even denser occupancy with agents.

Although solutions generated by the UniAGENT method are makespan suboptimal, they are obtained through optimization of a different parameter - namely the number of time layers in the ϑ-RELAXED encoding - hence their makespan is not as dramatically far from the optimum as in the case of rule based algorithms like PUSH-AND-ROTATE [de Wilde et al., 2014]. Altogether, UniAGENT solver represents a viable alternative to existing rule and search based CPF solvers.

References

[Ahuja *et al.*, 1993] **Ahuja**, R. K., **Magnanti**, T. L., **Orlin**, J. B. *Network flows: theory, algorithms, and applications.* Prentice Hall, 1993.

[Audemard and Simon, 2013] **Audemard**, G., **Simon**, L. *The Glucose SAT Solver.* http://labri.fr/perso/lsimon/ glucose/, 2013.

[Barer et al., 2014] **Barer**, M., **Sharon**, G., **Stern**, R., **Felner**, A. *Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem.* ECAI 2014 - 21st European Conference on Artificial Intelligence (ECAI 2014), pp. 961-962, IOS Press, 2014.

[Biere *et al.*, 2009] **Biere**, A., **Heule**, M., **van Maaren**, H., **Walsh**, T. *Handbook of Satisfiability.* IOS Press, 2009.

[Čáp *et al.*, 2013] **Čáp**, M., **Novák**, P., **Vokřínek**, J., **Pěchouček**, M. *Multi-agent RRT: sampling-based cooperative pathfinding.* International conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2013), pp. 1263-1264, IFAAMAS, 2013.

[Erdem *et al.*, 2013] **Erdem**, E., **Kisa**, D. G., **Öztok**, U., **Schüller**, P. *A General Formal Framework for Pathfinding Problems with Multiple Agents.* Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI 2013), AAAI Press, 2013.

[Gerevini *et al.*, 2008] **Gerevini**, A., **Saetti**, A., **Serina**, I., **Toninelli**, P. *LPG-td: fully automated planner for PDDL2.2 domains*, research web page, University of Brescia, Italy, http://zeus.ing.unibs.it/ lpg, 2008.

[Hsu *et al.*, 2006] **Hsu**, C. W., **Wah**, B. W., **Huang**, R., **Chen**, Y. X. *SGPlan 5: Subgoal Partitioning and Resolution in Planning*, research web page, University of Illinois, USA, http://wah.cse.cuhk.edu.hk/wah/programs/ SGPlan, 2006.

[Huang *et al.*, 2010] **Huang**, R., **Chen**, Y., **Zhang**, W. *A Novel Transition Based Encoding Scheme for Planning as Satisfiability.* Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010), AAAI Press, 2010.

[Kautz and Selman, 1999] **Kautz**, H., **Selman**, B. *Unifying SAT-based and Graph-based Planning.* Proceedings of the 16th International Joint Conference on Artificial Intelligence, pp. 318-325, Morgan Kaufmann, 1999.

[Kornhauser *et al.*, 1984] **Kornhauser**, D., **Miller**, G. L., **Spirakis**, P. G. *Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications.* Proceedings of the 25th Annual Symposium on Foundations of Computer Science (FOCS 1984), pp. 241-250, IEEE, 1984.

[Ratner and Warmuth, 1986] **Ratner**, D., **Warmuth**, M. K. *Finding a Shortest Solution for the $N \times N$ Extension of the 15-PUZZLE Is Intractable.* Proceedings of the 5th National Conference on Artificial Intelligence (AAAI 1986), pp. 168-172, Morgan Kaufmann, 1986.

[Ryan, 2008] **Ryan**, M. R. K. *Exploiting Subgraph Structure in Multi-Robot Path Planning.* Journal of Artificial Intelligence Research, Volume 31, pp. 497-542, AAA Press, 2008.

[Sharon et al., 2013] **Sharon**, G., **Stern**, R., **Goldenberg**, M., **Felner**, A. *The increasing cost tree search for optimal multi-agent pathfinding.* Artificial Intelligence, Volume 195, pp. 470-495, Elsevier, 2013.

[Silver, 2005] **Silver**, D. *Cooperative Pathfinding.* Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2005), pp. 117-122, AAAI Press, 2005.

[Standley and Korf, 2011] **Standley**, T. S., **Korf**, R. E. *Complete Algorithms for Cooperative Pathfinding Problems.* Proceedings of Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011), pp. 668-673, IJCAI/AAAI Press, 2011.

[Surynek, 2012a] **Surynek**, P. *Towards Optimal Cooperative Path Planning in Hard Setups through Satisfiability Solving*. Proceedings of 12th Pacific Rim International Conference on Artificial Intelligence (PRICAI 2012), pp. 564-576, Springer, 2012.

[Surynek, 2012b] **Surynek**, P. *On Propositional Encodings of Cooperative Path-Finding*. Proceedings of the 24th International Conference on Tools with Artificial Intelligence (ICTAI 2012), pp. 524-531, IEEE, 2012.

[Surynek, 2013] **Surynek**, P. *Mutex reasoning in cooperative path finding modeled as propositional satisfiability.* Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2013), pp. 4326-4331, IEEE, 2013.

[Surynek, 2014] **Surynek**, P. *Simple Direct Propositional Encoding of Cooperative Path Finding Simplified Yet More.* Proceedings of the 13th Mexican International Conference on Artificial Intelligence (MICAI 2014), pp. 410-425, Springer, 2014.

[Wagner and Choset, 2015] **Wagner**, G., **Choset**, H. *Subdimensional expansion for multirobot path planning.* Artificial Intelligence, Volume 219, pp. 1-24, Elsevier, 2015.

[Wehrle and Rintanen, 2007] **Wehrle**, M., **Rintanen**, J. *Planning as satisfiability with relaxed exist-step plans.* Proceedings of the 20th Australian Joint Conference on Artificial Intelligence, pp. 244–253, Springer, 2007.

[de Wilde et al., 2014] **de Wilde**, B., **ter Mors**, A. W., **Witteveen**, C. *Push and Rotate: a Complete Multi-agent Pathfinding Algorithm.* Journal of Artificial Intelligence Research, Volume 51, pp. 443-492, AAAI Press, 2014.

[Yu and LaValle, 2013] **Yu**, J., **LaValle**, S. M. *Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs.* Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI 2013), AAAI Press, 2013.

Pavel **Surynek**. *An Alternative Eager Encoding of the All-Different Constraint over Bit-Vectors.* Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012), pp. 927-928, Montpellier, France, IOS Press, 2012, ISBN 978-1-61499-097-0.

An Alternative Eager Encoding of the All-Different Constraint over Bit-Vectors

Pavel Surynek

Charles University Prague
Faculty of Mathematics and Physics
Department of Theoretical Computer Science and Mathematical Logic
Malostranské náměstí 25, Praha, 118 00, Czech Republic
pavel.surynek@mff.cuni.cz

Abstract. A novel eager encoding of the ALLDIFFERENT constraint over bit-vectors is presented in this short paper. It is based on 1 to 1 mapping of the input bit-vectors to a linearly ordered set of auxiliary bit-vectors. Experiments with four SAT solvers showed that the new encoding can be solved order of magnitudes faster than the standard encoding in a hard unsatisfiable case.

***Keywords*:** All-Different constraint, propositional encodings, SAT, linear ordering, bit-vectors

1. Introduction and Motivation

Models of many real-life problems require a subset of modeling variables to be pair-wise distinct. This requirement is known as an ALLDIFFERENT constraint 5 in the constraint programming context. As the SAT solving technology 1, 3, 6 is becoming a tool of choice in many practical applications, efficient manipulation with the ALLDIFFERENT constraint in SAT solvers is of interest. Unlike other works on translating the ALLDIFFERENT constraint into SAT that use direct encoding of variable's domains 4, we study how to encode the constraint over the set of bit-vectors which essentially use binary encoding. We present a new eager encoding that maps the given set of bit-vectors to a linearly ordered set of auxiliary bit-vectors. We show that the new encoding is more efficient for hard unsa-

tisfiable cases of the constraint on which SAT solvers struggle with the existent encoding for bit-vectors 2.

2. Background – Standard Model

Suppose to have a set of bit-vectors $\{\mathcal{B}^1, \mathcal{B}^2, \dots, \mathcal{B}^n\}$ each of length l. Bit-vectors are interpreted as non-negative integers. The ALLDIFFERENT constraint over $\mathcal{B}^1, \mathcal{B}^2, \dots, \mathcal{B}^n$ - denoted as ALLDIFFERENT($\{\mathcal{B}^1, \mathcal{B}^2, \dots, \mathcal{B}^n\}$) - requires that numbers represented by the bit-vectors are all distinct. The standard encoding 2 basically follows the scheme where pair-wise differences are encoded:

$$\text{ALLDIFFERENT}(\{\mathcal{B}^1, \mathcal{B}^2, \dots, \mathcal{B}^n\}) \equiv \bigwedge_{i,j=1 \wedge i<j}^{n} \mathcal{B}^i \neq \mathcal{B}^j$$

Trivially it is possible to encode the individual inequalities as follows. Let the $\mathbb{i}$-th bit of the k-th bit-vector with $\mathbb{i} \in \{1,2,\dots,l\}$ and $k \in \{1,2,\dots,n\}$ be denoted as $\mathcal{B}_{\mathbb{i}}^k$.

$$\mathcal{B}^i \neq \mathcal{B}^j \equiv \bigvee_{\mathbb{i}=1}^{l} \left(\neg\mathcal{B}_{\mathbb{i}}^i \vee \mathcal{B}_{\mathbb{i}}^j\right) \wedge \left(\mathcal{B}_{\mathbb{i}}^i \vee \neg\mathcal{B}_{\mathbb{i}}^j\right)$$

However, if unfolded into the CNF representation though the distributive rule it results into too many clauses which is impractical. Therefore encoding using auxiliary propositional variables is used. It follows the standard technique of Tseitin's hierarchical encoding. A fresh propositional variable is introduced for each inequality between individual bits of the involved bit-vectors. That is, there is a new variable $a_{\mathbb{i}}^{i,j}$ for every $i,j \in \{1,2,\dots,n\}$ with $i < j$ and $\mathbb{i} \in \{1,2,\dots,l\}$. The auxiliary variable indicates if the corresponding bits in the inequality between bit-vectors differ or not. Thus, the following clauses are included to express this interpretation:

$$\bigwedge_{\mathbb{i}=1}^{l} \left(\neg a_{\mathbb{i}}^{i,j} \vee \mathcal{B}_{\mathbb{i}}^i \vee \mathcal{B}_{\mathbb{i}}^j\right) \wedge \left(\neg a_{\mathbb{i}}^{i,j} \vee \neg\mathcal{B}_{\mathbb{i}}^i \vee \neg\mathcal{B}_{\mathbb{i}}^j\right)$$

Bit-vectors $\mathcal{B}^i$ and $\mathcal{B}^j$ differ if they differ in at least one position; that is, following clauses should be included: $\bigvee_{\mathbb{i}=1}^{l} a_{\mathbb{i}}^{i,j}$. Notice that auxiliary variables are linked to the original bits only in one direction. If $a_{\mathbb{i}}^{i,j}$ is set to $TRUE$ then $\mathcal{B}_{\mathbb{i}}^i$ and $\mathcal{B}_{\mathbb{i}}^j$ are forced to differ. However, if $a_{\mathbb{i}}^{i,j}$ is $FALSE$ then $\mathcal{B}_{\mathbb{i}}^i$ and $\mathcal{B}_{\mathbb{i}}^j$ are left unconstrained.

Proposition 1 (STANDARD ENCODING SIZE). The standard encoding of the ALLDIFFERENT constraint requires $l \cdot n$ propositonal variables to represent the

bit-vectors and $l \cdot \frac{n(n+1)}{2}$ auxiliary propositional variables; that is, $\mathcal{O}(l \cdot n^2)$ variables altogether. The number of clauses is $1 + l \cdot n(n+1)$; that is, $\mathcal{O}(l \cdot n^2)$. ■

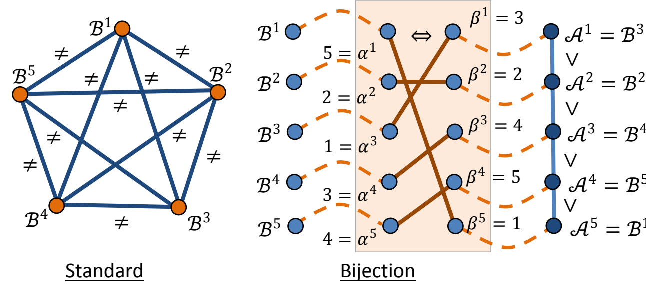


Figure 1. *Illustration of the standard and the bijection* ALLDIFFERENT *encodings.* In the bijection encoding, a 1-to-1 mapping of the bit-vectors is found first. Then the values of bit-vectors are forced to be linearly ordered according to their position in the mapping.

3. Alternative Bijection Encoding

We observed that a SAT solver struggles over the standard encoding especially in the unsatisfiable case according to our preliminary experiments. Therefore we developed an alternative encoding that is more suitable for this case. It maps the original bit-vectors to a linearly ordered set of auxiliary bit-vectors. First, a 1-to-1 mapping (*bijection*) between sets of bit-vectors needs to be modeled to enable this encoding style (see
Figure 1 for illustration).

Let the new linearly ordered bit-vectors be denoted as $\mathcal{A}^1, \mathcal{A}^2, \dots, \mathcal{A}^n$. Additionally bit-vectors $\alpha^1, \alpha^2, \dots, \alpha^n$ and $\beta^1, \beta^2, \dots, \beta^n$ of size $\lceil \log_2 n \rceil$ are introduced to model the bijection. The bit-vector α^k indicates what $\mathcal{A}^i$ with $i \in \{1,2,\dots,n\}$ the original $\mathcal{B}^k$ will be mapped to. Bit-vectors β^j are used to enforce that at most one original bit-vector is mapped to a single ordered bit-vector. The following integer constraints are to establish the bijection:

$$\bigwedge_{i,k=1}^{n} \alpha^k = i \implies \mathcal{B}^k = \mathcal{A}^i \wedge \beta^i = k$$

It is crucial, that domains of bit-vectors α^k and β^j consists of exactly n values to ensure that the bijection is modeled correctly (extra values are forbidden). The individual integer implication is encoded with a single auxiliary propositional variable e_i^k as follows:

$$e_i^k \vee \bigvee_{\mathbb{i}=1}^{\lceil \log_2 n \rceil} \neg \mathrm{lit}(i_{\mathbb{i}}, \alpha_{\mathbb{i}}^k) \quad \text{where} \quad \mathrm{lit}(i_{\mathbb{i}}, \alpha_{\mathbb{i}}^k) = \begin{cases} \alpha_{\mathbb{i}}^k & \text{iff } i_{\mathbb{i}} = 1 \\ \neg \alpha_{\mathbb{i}}^k & \text{iff } i_{\mathbb{i}} = 0 \end{cases}$$

$$\bigwedge_{\mathbb{i}=1}^{l} \left(\neg e_i^k \vee \neg \mathcal{B}_{\mathbb{i}}^k \vee \mathcal{A}_{\mathbb{i}}^i\right) \wedge \left(\neg e_i^k \vee \mathcal{B}_{\mathbb{i}}^k \vee \neg \mathcal{A}_{\mathbb{i}}^i\right)$$

$$\bigwedge_{\mathbb{i}=1}^{\lceil \log_2 n \rceil} \neg e_i^k \vee \mathrm{lit}(k_{\mathbb{i}}, \beta_{\mathbb{i}}^i)$$

Finally, there are integer constraints enforcing the ordering:

$$\bigwedge_{i=1}^{n-1} \mathcal{A}^i < \mathcal{A}^{i+1}$$

The individual inequality is encoded as a strict lexicographic ordering over the two bit-vectors. Now, l fresh propositional variables $\mathcal{f}_{\mathbb{i}}^i$ with $\mathbb{i} \in \{1,2,\dots,l\}$ are introduced to indicate the first bit where $\mathcal{A}^i$ is less than $\mathcal{A}^{i+1}$. The ordering itself then just means that there exists such a first bit where bit-vectors differ: $\bigvee_{\mathbb{i}=1}^{l} \mathcal{f}_{\mathbb{i}}^i$.

$$\bigwedge_{\mathbb{j}=1}^{\mathbb{i}-1} \left(\neg \mathcal{f}_{\mathbb{i}}^i \vee \neg \mathcal{A}_{\mathbb{j}}^i \vee \mathcal{A}_{\mathbb{j}}^{i+1}\right) \wedge \left(\neg \mathcal{f}_{\mathbb{i}}^i \vee \mathcal{A}_{\mathbb{j}}^i \vee \neg \mathcal{A}_{\mathbb{j}}^{i+1}\right)$$

$$\left(\neg \mathcal{f}_{\mathbb{i}}^i \vee \neg \mathcal{A}_{\mathbb{i}}^i\right) \wedge \left(\mathcal{A}_{\mathbb{i}}^{i+1} \vee \neg \mathcal{f}_{\mathbb{i}}^i\right)$$

Proposition 2 (BIJECTION ENCODING SIZE). The bijection encoding requires $2l \cdot n$ propositional variables to represent the bit-vectors, $2n\lceil log_2 n \rceil$ variables to represent the bijection, and $n^2 + l(n-1)$ auxiliary propositional variables; that is $O(n \cdot max\{n, l\})$ propositional variables altogether.

The number of clauses is $n^2(1 + l + \lceil \log_2 n \rceil) + (n-1)\frac{2l(l+1)}{2}$; *that is,* $O(n^2 \cdot max\{\lceil \log_2 n \rceil, l\} + n \cdot l^2)$. ■

Table 1. Comparison of sizes of the standard and the bijection encoding.

#bit-vectors (16-bits)	Standard		Bijection	
	#Variables	#Clauses	#Variables	#Clauses
64	67584	133056	9968	176943
128	266240	536448	28400	690031
256	1056768	2154240	90096	2756591

4. Experimental Evaluation

As shown in Table 1, the bijection encoding has fewer variables while the number of clauses is slightly higher than in the standard encoding. Nevertheless, we also need runtime comparison. A setup where a *transition-phase* behavior was observed is presented. We used 32 bit-vectors consisting of 6 bits. Additionally, there was a lower bound and an upper bound per each bit-vector. If $d \in \mathbb{N}, d \leq$

34 is a given domain size, then the lower bound $b_L^k \in \mathbb{N}$ and the upper bound $b_U^k \in \mathbb{N}$ for the bit-vector $\mathcal{B}^k$ were generated randomly as follows: b_L^k was selected uniformly from $[0..34-d]$ and b_U was set to $b_U^k + d$. Thus, $b_L^k \leq \mathcal{B}^k \leq b_U^k$ is enforced for each k. Finally, a single ALLDIFFERENT over 32 bit-vectors was added.

Three SAT solvers were used in the evaluation: MINISAT 3, GLUCOSE 1, and CRYPTOMINISAT 6. The runtime was measured for different domain sizes d ranging from 2 to 34 - Figure 2. For small d unsatisfiability could be checked easily; for large d the same could be done for satisfiability. The most interesting behavior occurred around $d = 13$ which represent difficult cases.

None of the tested SAT solvers was able to solve all the instances over the standard encoding in the time limit of 1 hour (wall clock limit per instance). The best performing over the standard encoding was GLUCOSE which solved 29 instances out of 33 and was also the fastest. Over the bijection encoding, MINISAT and CRYPTOMINISAT solved all the instances and very importantly the runtime of CRYPTOMINISAT was always below 2 seconds. GLUCOSE also performed relatively well compared to the standard encoding with 30 solved instances.

Generally, the standard encoding can be solved faster in the satisfiable case. However, the bijection encoding is significantly better in the hard unsatisfiable case. This is because it can be checked more easily for this encoding if there are enough values in domains of bit-vectors to establish the required pair-wise difference (at least by some SAT solvers). A single linearly ordered set of bit-vectors is matched into the domains while in case of the standard encoding all the orderings (permutations) of the original bit-vectors may be checked.

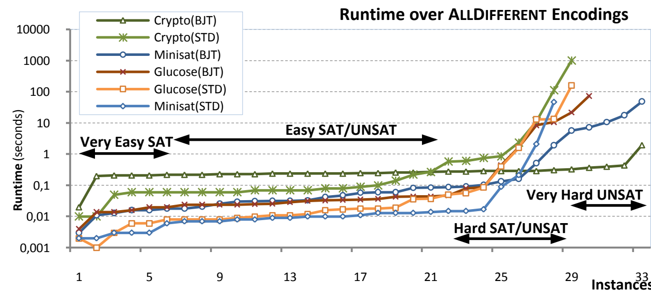


Figure 2. Instances are sorted according to the increasing runtime.

5. Conclusions

A new encoding for the ALLDIFFERENT constraint over bit-vectors based on 1-to-1 mapping has been proposed. It has fewer variables and it is more efficient in difficult unsatisfiable cases than the existent encoding 2 that uses pair-wise differences. In the future work, it would be also interesting to investigate how the presented eager encodings performs with respect to the strong ALLDIFFERENT propagators 5 integrated with the solver lazily via the SMT framework and also how it performs in applications.

References

1. G. **Audemard**, L. **Simon**, 'Predicting Learnt Clauses Quality in Modern SAT Solver', Proceedings of IJCAI 2009, (2009).
2. A. **Biere**, R. **Brummayer**, 'Consistency Checking of All Different Constraints over Bit-Vectors within a SAT Solver', Proceedings of FMCAD 2008, 1-4, (2008).
3. N. **Eén**, N. **Sörensson**, 'An Extensible SAT-solver', Proceedings of SAT 2003, 502-518, (2003).
4. P. **Nightingale**, I. **Gent**, 'A New Encoding of AllDifferent into SAT', CP 2004 Workshop on Modelling and Reformulating CSPs, (2004).
5. J.-C. **Régin**, 'A Filtering Algorithm for Constraints of Difference in CSPs', Proceedings of AAAI 1994, 362-367, (1994).
6. M. **Soos**, K. **Nohl**, C. **Castelluccia**, 'Extending SAT Solvers to Cryptographic Problems', Proceedings of SAT 2009, 244-257, (2009).

Articles concerning recent research

An Improved Sub-optimal Algorithm for Solving $(N^2 - 1)$-Puzzle and Applications in Cooperative Path-Finding

Pavel Surynek[1] and **Petr Michalík**[2]

[1]*Charles University in Prague*
Faculty of Mathematics and Physics
Department of Theoretical Computer Science and Mathematical Logic
Malostranské náměstí 25, Praha, 118 00, Czech Republic

[2]*Accenture Central Europe B.V. Consulting*
Jiráskovo náměstí 1981/6, Praha, 120 00, Czech Republic

pavel.surynek@mff.cuni.cz, petr.michalik@accenture.com

Abstract. The problem of solving $(n^2 - 1)$-puzzle and cooperative path-finding (CPF) sub-optimally is addressed in this manuscript. The task in the puzzle is to rearrange $n^2 - 1$ pebbles on the square grid of the size of $n \times n$ using one vacant position to a desired goal configuration. An improvement to the existent polynomial-time algorithm is proposed and experimentally analyzed. The improved algorithm is trying to move pebbles in more efficient way than the original one by grouping them into so-called snakes and moving them jointly within the snake. An experimental evaluation showed that the algorithm using snakes produces solutions that are 8% to 9% shorter than solutions generated by the original algorithm.

The snake-based relocation was also integrated into an algorithm for solving the CPF problem sub-optimally, which is a closely related task. The task in CPF is to relocate a group of abstract robots that move over an undirected graph to given goal vertices. Robots can move to unoccupied neighboring vertices and at most one robot can be placed in each vertex. The $(n^2 - 1)$-puzzle is a special case of CPF where the underlying graph is represented by a 4-connected grid and there is only one vacant vertex. Improvements gained by using snakes in the algorithm for CPF were around 30% in $(n^2 - 1)$-puzzle solving and up to 50% in CPFs over biconnected graphs with various ear decompositions and multiple vacant vertices.

***Keywords*:** $(n^2 - 1)$-puzzle, 15-puzzle, cooperative path-finding, polynomial complexity, multi-robot path planning, algorithm *BIBOX*

1. Introduction and Motivation

The (n^2-1)-puzzle [2, 3, 5, 6] represents one of the best-known examples of a so-called *cooperative path-finding* (CPF) [12, 16, 19, 23] problem. It is important both practically and theoretically. From the theoretical point of view it is interesting for the hardness of its optimization variant which is known to be NP-hard [7, 8].

Practically it is important since many real-life relocation problems can be solved by techniques developed for the (n^2-1)-puzzle. Those include *path planning for multiple robots* [10, 11, 13, 14, 17, 20, 23], *rearranging* of shipping containers in warehouses, or *coordination* of vehicles in dense traffic. Moreover, the reasoning about relocation/coordination tasks should not be limited to physical entities only. Many tasks such as planning of *data transfer*, *commodity transportation*, and *motion planning* of units in *computer-generated imagery* can be tackled using techniques originally developed for the (n^2-1)-puzzle.

In this manuscript, we concentrate ourselves on solving the (n^2-1)-puzzle sub-optimally, that is, by a fast polynomial-time algorithm. We are trying to improve the basic incremental placing of pebbles as it is done by the existent on-line solving algorithm of Parberry [6] by moving them in groups called *snakes*. Moving pebbles jointly in snakes is supposed to be more efficient in terms of the total number of moves than moving them individually as it was originally proposed [6]. An improved algorithm exploiting snake-based movements is presented.

We utilized experiences gained during making snake-based improvements to Parberry's algorithm in solving CPF. We observed that existent *BIBOX* algorithm [13, 16] for solving CPF sub-optimally over *biconnected graphs* with at least two vacant vertices operates in a similar way to Parberry's algorithm and hence snake-based reasoning can be integrated into it.

An extensive competitive experimental evaluation was done to evaluate qualities of snake-based improvements in solving (n^2-1)-puzzle as well as in solving CPFs over biconnected graphs.

The manuscript is organized as follows. The problem of (n^2-1)-puzzle is formally introduced in Section 2. An overview of existent solving algorithm and other related solving approaches is given in Section 3. The main part of the paper is constituted by Section 4 and Section 5 where the *snake movement* is introduced into Parberry's algorithm and into the *BIBOX* algorithm for CPF solving. An extensive experimental evaluation is given in Section 6.

2. Problem Statement

The (n^2-1)-puzzle consists of a set of pebbles that are moved over a square grid of size $n \times n$ [1, 6, 7, 8, 24]. There is exactly one position vacant on the grid and others are occupied by exactly one pebble. A pebble can be moved to the adjacent vacant position. The task is to rearrange pebbles on the grid into a desired goal state.

2.1. *Formal Definition*

Sets of pebbles will be denoted as Ω_n for $n \in \mathbb{N}$. It holds that $|\Omega_n| = n^2 - 1$ for every $n \in \mathbb{N}$. It is supposed that pebbles from a set Ω_n are arranged on a square grid of the size $n \times n$ where each pebble is placed into one of the cells of the grid. There is at most one pebble in each cell of the grid; one cell on the grid remains always vacant (Figure 1).

Definition 1 ***(configuration in a grid).*** An *configuration* of a set of pebbles Ω_n in a square grid of the size $n \times n$ with $n \in \mathbb{N}$ is fully described using two functions $x_n: \Omega_n \longrightarrow \mathbb{N}$ and $y_n: \Omega_n \longrightarrow \mathbb{N}$ that satisfy the following *puzzle conditions*:

(i) $x_n(p) \in \{1,2,\ldots,n\}$ and $y_n(p) \in \{1,2,\ldots,n\}$ $\forall p \in \Omega_n$ (1)

(ii) $|\{p \in \Omega_n | (x_n(p), y_n(p)) = (i,j)\}| \leq 1$ for $\forall i,j \in \{1,2,\ldots,n\}$ (2)
(every cell of the grid is occupied by at most one pebble)

(iii) $\exists i,j \in \{1,2,\ldots,n\}$ such that $\forall p \in \Omega_n$ $(x_n(p), y_n(p)) \neq (i,j)$ (3)
(there exists a cell in the grid that remains vacant).

For convenience, we will also use some kind of an inverse to x_n and y_n which will be called an *occupancy* function and denoted as $\sigma_n: \{1,2,\ldots,n\} \times \{1,2,\ldots,n\} \longrightarrow \Omega_n \cup \{\bot\}$. It holds that $\sigma_n(i,j) = p$ if and only if $\exists p \in \Omega_n$ such that $x_n(p) = i$ and $y_n(p) = j$ or $\sigma_n(i,j) = \bot$ if no such pebble p exists (that is, if the cell (i,j) is vacant). □

The configuration of pebbles in the grid can be changed through moves. An allowed *move* is to shift a pebble horizontally or vertically from its original cell to the adjacent vacant cell. Formally, the notion of move is described in the following definition. Four types of moves are distinguished here: *left*, *right*, *up*, and *down* – only left move is defined formally; *right*, *up*, and *down* moves are analogous.

Definition 2 ***(left move).*** A *left* move with pebble $p \in \Omega_n$ can be done if $x_n(p) > 1$ and $\sigma_n(x_n(p)-1, y_n(p)) = \perp$; it holds for the resulting configuration after the move described by $x_n^{'}$ and $y_n^{'}$ that $x_n^{'}(q) = x_n(q)$ and $y_n^{'}(q) = y_n(q)$ $\forall q \in \Omega_n$ such that $q \neq p$ and $x_n^{'}(p) = x_n(p) - 1$ and $y_n^{'}(p) = y_n(p)$. □

We are now able to define the (n^2-1)-puzzle using the formal constructs we have just introduced. The task is to transform a given initial configuration of pebbles in the grid to a given goal one using a sequence of allowed moves.

Definition 3 *(*$(\boldsymbol{n^2-1})$**-puzzle***).* An *instance of the* (n^2-1)*-puzzle* is a tuple $(n, \Omega_n, x_n^0, y_n^0, x_n^+, y_n^+)$ where $n \in \mathbb{N}$ is the size of the instance, Ω_n is a set of pebbles, x_n^0 and y_n^0 is a pair of functions that describes the *initial configuration* of pebbles in the grid, and x_n^+ and y_n^+ is a pair of functions that describes the *goal configuration* of pebbles. The task is to find a sequence of allowed moves that transforms the initial configuration into the goal one. Such sequence of moves will be called a *solution* to the instance. □

Again it is supposed that the occupancy function is available with respect to the initial configuration x_n^0, y_n^0 and the goal configuration x_n^+, y_n^+; that is, we are provided with occupancy functions σ_n^0 and σ_n^+. To avoid special cases it will be also supposed that $\sigma_n^+(n,n) = \perp$; that is, the vacant position is finally in the right bottom corner.

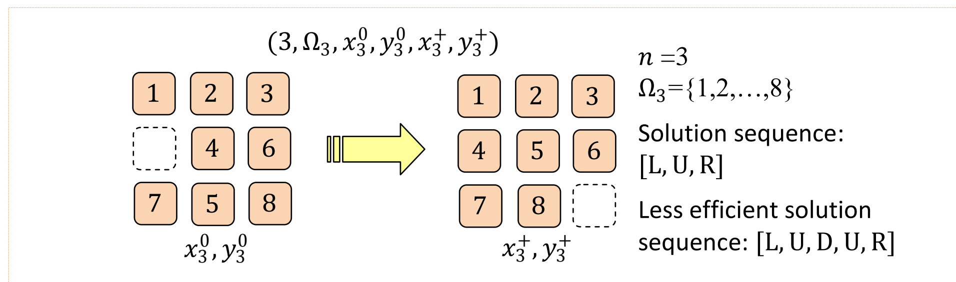


Figure 1. *An illustration of the* (n^2-1)*-puzzle.* The initial and the goal configuration of pebbles on the square grid of size 3 × 3 are shown. Two solutions of the instance are shown as well.

2.2. *Complexity and Variants of the Problem*

It is known that the decision variant of the (n^2-1)-puzzle (that is, the yes/no question whether there exists a solution to the given instance) is in P [1, 6, 24]. It can be checked by using simple parity criterion. Using techniques for rearranging

pebbles over graphs [1] a solution of length $\mathcal{O}(n^6)$ can be constructed in the worst-case time of $\mathcal{O}(n^6)$ if there exists any. An approach dedicated exclusively to the $(n^2 - 1)$-puzzle is [6] able to generate a solution of length $\mathcal{O}(n^3)$ in the worst-case time of (n^3) if there exists any.

If a requirement on the length of the solution is added, the problem becomes harder. It is known that the decision problem of whether there exists a solution to a given $(n^2 - 1)$-puzzle of at most the given length is NP-complete [8].

3. The Original Solving Algorithm and Related Works

A special sub-optimal solving algorithm dedicated for the $(n^2 - 1)$-puzzle has been proposed by Parberry in [6]. As our new solving algorithm is based on the framework of the original one, we need to recall it at least briefly in this section.

3.1. *Algorithm of Parberry*

The algorithm of Parberry [6] sequentially places pebbles into rows and columns. More precisely, pebbles are placed sequentially into the first row and then into the first column, which reduces the instance to that of the same type but smaller – that is, we obtain an instance of the $((n-1)^2 - 1)$-puzzle.

Algorithm 1. *The original algorithm of Parberry for solving the* $(n^2 - 1)$*-puzzle* [6]. The main loop of the algorithm is shown. Detailed description of placement of individual pebbles is not shown here –it will be discussed in the context of new approach for pebble placement.

procedure *Solve-N^2-1-Puzzle*$(n, \Omega_n, x_n^0, y_n^0, x_n^+, y_n^+)$
/* A procedure that produces a sequence of moves that solves the given $(n^2 - 1)$-*puzzle*.
Parameters: n, Ω_n - a size of the puzzle and a set of pebbles,
x_n^0, y_n^0 - an initial configuration of pebbles in the grid,
x_n^+, y_n^+ - a goal configuration of pebbles in the grid. */

1: $(x_n, y_n) \leftarrow (x_n^0, y_n^0)$
2: for $i = 1,2, \dots, n-3$ do
3: for $j = i, i+1, \dots, n$ do {current row is solved – from the left to the right}
4: $p \leftarrow \sigma_n^+(i,j)$
5: if $(i,j) \neq (x_n^+(p), y_n^+(p))$ then
6: $(x_n, y_n) \leftarrow$ *Place-Pebble*(x_n, y_n, i, j, p)
7: $\Omega_n \leftarrow \Omega_n \setminus \{p\}$
8: for $j = n, n-1, \dots, i+1$ do {current column is solved – from the bottom to the up}
9: $p \leftarrow \sigma_n^+(i,j)$
10: if $(i,j) \neq (x_n^+(p), y_n^+(p))$ then
11: $(x_n, y_n) \leftarrow$ *Place-Pebble*(x_n, y_n, i, j, p)
12: $\Omega_n \leftarrow \Omega_n \setminus \{p\}$
13: $\Omega_3 \leftarrow \Omega_n$; $x_3^0 \leftarrow x_n|_{\Omega_3}$; $y_3^0 \leftarrow y_n|_{\Omega_3}$; $x_3^+ \leftarrow x_n^+|_{\Omega_3}$; $y_3^+ \leftarrow y_n^+|_{\Omega_3}$ {restriction on Ω_n}
14: *Solve-8-Puzzle*$(\Omega_3, x_3^0, y_3^0, x_3^+, y_3^+)$ {the residual 8-puzzle is solved by A* algorithm}

This process of pebble placement is repeated until an 8-puzzle on the grid of size 3×3 is obtained. The final case of the 8-puzzle is then solved optimally by the A* algorithm [9].

The main loop of the algorithm is shown in pseudo-code as Algorithm 1. The algorithm uses two high-level functions *Place-Pebble*, which conducts placement of a pebble to a given position, and *Solve-8-Puzzle*, which finalizes the solution by solving the residual 8-puzzle.

The placement of pebbles implemented within the function *Place-Pebble* will be discussed in more details later in the context of our improvement. Nevertheless, it is done quite naturally by moving a pebble first *diagonally* towards the goal position if necessary and then *horizontally* or *vertically*. To be able to conduct diagonal, horizontal and vertical movement a vacant position needs to be moved together with the pebble being placed. Actually, the vacant position is moving around the pebble always to the front in the direction of the intended move. After having vacant position in the front, the pebble is moved forward. It is necessary to avoid already placed pebbles when placing a new one.

3.2. ***Other Related Works***

The (n^2-1)-puzzle represents a special variant of a more general problem of *cooperative path-finding* - CPF (also known as *pebble motion problem on a graph*) [3, 4, 10, 11, 12, 14, 24]. The generalization consists in the fact that there is an arbitrary undirected graph representing the environment instead of the regular 4-connected grids as it is in the case of (n^2-1)-puzzle. There are also pebbles, in context of CPF called *robots* (or agents), that are placed in vertices of the graph while at least one vertex remains vacant. The allowed state transition is a single move with a robot to a vacant adjacent vertex. The task is expectably to rearrange robots from a given initial configuration to a given goal one.

Although the problem has been studied long time ago [3, 24] recently there has been a considerable progress. The first new work showing solvability of every instance of pebble motion problem consisting of biconnected graph [18, 21, 22] containing at least two vacant positions is [13]. The related solving algorithm called *BIBOX* [13] can produce solution of length at most $\mathcal{O}(|V|^3)$ in the worst-case time of $\mathcal{O}(|V|^3)$ (V is the set of vertices of the input graph). The *BIBOX* algorithm also generates solutions that are significantly shorter than those generated by algorithms from previous works are [3, 24].

More results followed then. A generalization of *BIBOX* algorithm called *BIBOX-θ* is described in [14]. It does not need the second vacant position and again can solve instances on bi-connected graphs (notice that the grid of the (n^2-1)-

puzzle is a bi-connected graph; hence *BIBOX-θ* is applicable to it). Theoretically, it generates solutions of the worst-case length of $O(|V|^4)$; however, practically solutions are much shorter.

Two years later an algorithm called *PUSH-and-SWAP* has been published in [4] – it shows that for every solvable instance on arbitrary graph containing at least two vacant positions a solution of length $O(|V|^3)$ can be generated. The algorithm however contains some errors and the its correction *PUSH-and-ROTATE* has been published later in [23].

In all the above results the solution length is sub-optimal and the worst-case time complexity is guaranteed (it is polynomial). A progress has been also made in optimal solving of the pebble motion problem. A new technique that can optimally solve a special case consisting of a grid with obstacles and relatively small number of pebbles is described in [17]. It is based on an informed search, which however does not guarantee time necessary to produce a solution (the time may be exponential in the size of the instance).

Special cases of the problem with large graphs and relatively sparsely arranged pebbles are studied in [19, 20]. These new techniques are focused on applications in *computer games*. The complexity as well as the solution quality is guaranteed by these techniques. Another specialized technique for relatively large graphs and small number of pebbles has been developed within [10, 11]. The graph representing the environment is decomposed into subgraph patterns, which are subsequently used for more efficient solving by search.

4. A New Solving Approach Based on 'Snakes'

In this section, we are about to define a new concept of a so-called *snake*. Informally, a snake is a sequence of pebbles that consecutively neighbors with a pebble that proceeds. As we will show, moving and placing a snake as a whole is much more efficient than moving and placing individual pebbles it consists of.

Recall that original algorithm for solving the puzzle [6] places pebbles individually into currently solved row or column. This may be inefficient if two or more pebbles that need to be placed are grouped together in some location distant from their goal location. In such a case, it is necessary that the vacant position is moved together with the pebble being placed and then it is moved back to the distant location to allow movement of the next pebble. If we manage all the pebbles forming the group to move from their distant location to their goal positions jointly, multiple movements of the vacant position between the distant location and goal positions may be eliminated.

4.1. ***Formal Definition of a 'Snake'***

Consider a situation shown in Figure 2 where pebbles 1 and 2 are grouped together in a location distant from their goal positions. The original algorithm consumes $16n-20$ moves to place both pebbles successfully to their goal positions. If pebbles are moved not one by one but jointly as it is shown in Figure 3, much less movements are necessary. Grouping pebbles can save up to $4n$ moves.

This is the basic idea behind the concept of snake. Let us start with definition of a *metric* on the grid of the puzzle. Then the definition of the snake will follow.

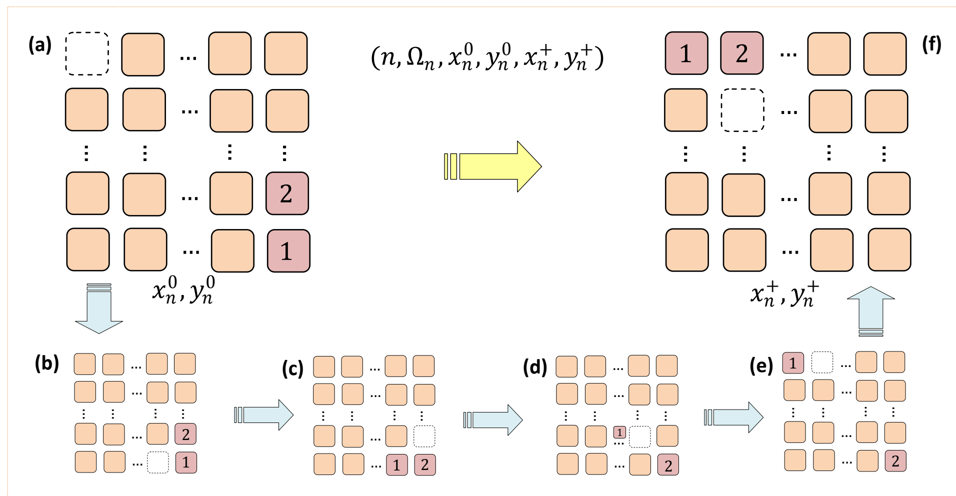


Figure 2. *A setup of the* (n^2-1)*-puzzle where the original algorithm* [6] *is inefficient.* Pebbles 1 and 2 need to be moved from the bottom right corner (a) to the upper left corner (f). First, pebble 1 is moved diagonally to its goal position (b, c, d, and e). After pebble 1 is successfully placed, vacant position is moved towards pebble 2 and it starts to move in the same way as pebble 1 to its goal position. The whole process of rearranging consumes $16n-20$ moves.

Definition 4 ***(Manhattan distance).*** A *Manhattan distance* for the (n^2-1)-puzzle $\mu_n: \{1,2,\ldots,n\}^2 \times \{1,2,\ldots,n\}^2 \longrightarrow \{0,1,\ldots,2n-1\}$ is a metric on the square grid such that $\mu_n((x_1,y_1);(x_2,y_2)) = |x_1-x_2| + |y_1-y_2|$. □

The input parameters of the Manhattan distance μ are coordinates of two positions. Having a metric on the grid of the puzzle, we are able to define neighborhood of a pebble. A snake will be then defined using the notion of neighborhood as a sequence of pebbles that consecutively lies in neighborhood of a pebble that proceeds.

Definition 5 ***(Manhattan neighborhood).*** A *Manhattan neighborhood* of a pebble p denoted as $\nu(p)$ is a set of those pebbles that are located directly left, right, above and below to p with respect to the configuration on the grid. That is, $\nu(p) = \{q \in \Omega_n | \mu_n((x_n(p), y_n(p)); ((q), y_n(q))) = 1\}$. □

Definition 6 ***(Snake).*** A *snake* s of size k is a sequence of pebbles $s = [s_1, s_2, \ldots, s_k]$ such that $\forall i \in \{1,2,\ldots,k\}$ $s_i \in \Omega_n$ and $\forall j \in \{2,3,\ldots,k\}$ $s_j \in \nu(s_{j-1})$. Pebble s_1 is called a *head* of the snake; pebble s_k is called a *tail* of the snake. □

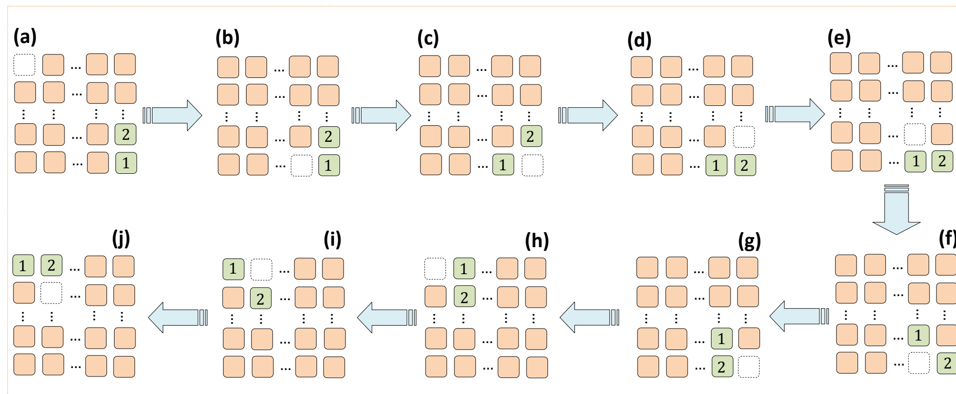


Figure 3. *Placing grouped pebbles using a snake.* The situation from Figure 2 is solved by grouping pebbles 1 and 2 into a snake, which is then moved as a whole from its original location in bottom right corner to the goal position in the upper left corner. The process consumes $12n + \mathcal{O}(1)$ which is approximately $4n$ better than the original approach that places pebbles individually.

Notice that each pebble itself forms a trivial snake of size 1. Composed movements of a snake *horizontally*, *vertically*, and *diagonally* can be defined analogically as in the case of a single pebble. If fact, they are generalizations of composed movements for single pebble. It is always assumed that the vacant position is in front of the head of snake in the direction of the intended movement. In such a setup, the snake can move forward by one position. The vacant position then needs to be moved around the snake in front of its head again to allow the next movement forward. See Figure 4 for illustration of composed movements for snakes (movements for a snake of length 2 are shown; it is easy to generalize composed movements for snakes of arbitrary length).

The horizontal and vertical composed movements consume $2k + 3$ moves. The number of moves consumed by the diagonal movement depends on the shape of a snake in the middle section – it is not that easy to express. However, if we

need to move a snake of length 2 diagonally forward following the shape from Figure 4, then it consumes 10 moves.

Unfortunately it is rarely the case that a group of pebbles in some distant from goal location forms a snake. Even it is not that frequent that pebbles which are to be placed consecutively are close to each other. Hence, to take the advantage of moving a group of pebbles as a snake we need first to form a snake of them. This is however not for free as a number of moves are necessary to form a snake. Thus, it is advisable to consider whether forming a snake is worthwhile. Moreover, there are many ways how to form a snake while each may be of different cost in terms of the number of moves.

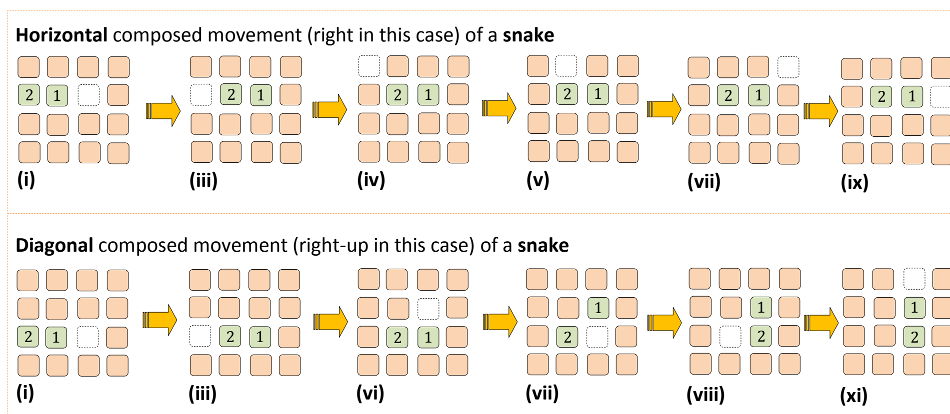


Figure 4. *Composed movements of a snake of length 2.* The horizontal and diagonal composed movements of a snake of length 2 are shown. Other cases as well as generalization for snakes of arbitrary length are straightforward.

Generally, the simplest way is to move one pebble to the other or vice versa in order to form a snake of length 2. It is known by using above calculations what number of moves is consumed by moving a snake as well as what number of moves are consumed by moving a pebble towards other pebble. Hence, it is easy to estimate the cost of using a snake in either of both ways as well as the cost of not using it at all in terms of the number of moves. Thus, it is possible to choose the most efficient option. This is another core idea of our new algorithm.

4.2. *A 'Snake' Based Algorithm*

Our new algorithm for solving the $(n^2 - 1)$-puzzle will use snakes of length 2. The algorithm proceeds in the same way as the original algorithm of Parberry [6]. That is, pebbles are placed into the first row and then into the first column and

after the first row and the first column are finished the task is reduced to the puzzle of the same type but smaller (namely, the task is reduced to solve the $((n-1)^2-1)$-puzzle). The trivial case of the 8-puzzle on a grid of the size 3×3 is again solved by the A* algorithm [9].

Algorithm 2. *The main function of a new algorithm for solving the* (n^2-1)*-puzzle.* The function for producing a sequence of moves for placing two consecutive pebbles using snakes (if using snakes turns out to be beneficial) is shown.

function *Place-Pebbles*$(x_n, y_n, x_n^+, y_n^+, p, q)$: **pair**
/* A function that produces a sequence of moves for placing two consecutive pebbles with respect to the order of placement. The new configuration is returned in a return value.
Parameters: x_n, x_y - a current configuration of pebbles in the grid,
x_n^+, y_n^+ - a goal configuration of pebbles in the grid,
p, q - two consecutive pebbles that will be placed. */

1: $c \leftarrow \text{cost}_1(x_n, y_n, x_n^+, y_n^+)(p, q)$
2: $e_{p,q} \leftarrow \text{estimate}_{\text{snake}}(x_n, y_n, x_n^+, y_n^+)(p, q)$
3: $e_{q,p} \leftarrow \text{estimate}_{\text{snake}}(x_n, y_n, x_n^+, y_n^+)(q, p)$
4: **if** $\min\{e_{p,q}, e_{q,p}\} < 1.2c$ **then**
5: **if** $e_{p,q} < e_{q,p}$ **then**
6: **let** (i,j) be a position such that $|i - x_n(p)| + |j - y_n(p)| = 1$
7: $(x_n, y_n) \leftarrow$ *Move-Vacant*(x_n, y_n, i, j)
8: $d_{min} \leftarrow \min\{|i' - x_n(p)| + |j' - y_n(p)| \mid i', j' \in \mathbb{N} \wedge |i' - x_n(q)| + |j' - y_n(q)| = 1\}$
9: **let** (i,j) be a position such that $|i - x_n(p)| + |j - y_n(p)| = d_{min}$
10: $(x_n, y_n) \leftarrow$ *Place-Pebble*(x_n, y_n, i, j, p)
11: **else**
12: **let** (i,j) be a position such that $|i - x_n(q)| + |j - y_n(q)| = 1$
13: $(x_n, y_n) \leftarrow$ *Move-Vacant*(x_n, y_n, i, j)
14: $d_{min} \leftarrow \min\{|i' - x_n(q)| + |j' - y_n(q)| \mid i', j' \in \mathbb{N} \wedge |i' - x_n(p)| + |j' - y_n(p)| = 1\}$
15: **let** (i,j) be a position such that $|i - x_n(q)| + |j - y_n(q)| = d_{min}$
16: $(x_n, y_n) \leftarrow$ *Place-Pebble*(x_n, y_n, i, j, q)
17: **let** $s = [p, q]$ be a snake {actually p and q form a snake at this point}
18: **let** π be a shortest path from $(x_n(p), y_n(p))$ to $(x_n^+(p), y_n^+(p))$ such that $\pi[|\pi| - 1] = (x_n^+(q), y_n^+(q))$ and π does not intersect any position containing already placed pebble
19: **for** $k = 1, 2, \ldots, |\pi| - 1$ **do**
20: $(x_n, y_n) \leftarrow$ *Snake-Composed-Movement*$(x_n, y_n, \pi[k], \pi[k+1], s)$
{when vacant position is moved it should avoid already placed pebbles}
21: **else**
22: $(x_n, y_n) \leftarrow$ *Place-Pebble*$(x_n, y_n, x_n^+, y_n^+, p)$
23: $(x_n, y_n) \leftarrow$ *Place-Pebble*$(x_n, y_n, x_n^+, y_n^+, q)$
24: **return** (x_n, y_n)

Along the solving process, the concept of snakes is used to move pebbles in a more efficient way. The basic idea is to make an estimation whether it will be beneficial to form a snake of two pebbles that are about to be placed. If so then a snake is formed in one of the two ways – the first pebble is moved towards the

second one or vice versa – the better option according to the estimations is always chosen. If forming a snake turns out not to be beneficial then pebbles are moved in the same way as in the case of the original algorithm; that is, one by one.

Let $\text{estimate}_{\text{snake}}\,(x_n, y_n, x_n^+, y_n^+): \Omega_n \times \Omega_n \longrightarrow \mathbb{N}_0$ is a functional that estimates the number of moves necessary to place a given two pebbles using the snake like motion. More precisely, $\text{estimate}_{\text{snake}}\,(x_n, y_n, x_n^+, y_n^+)(p, q)$ is the estimation of the number of moves necessary to form a snake by moving pebble p towards q and to place the formed snake into the goal location where x_n, y_n and x_n^+, y_n^+ denote the current and the goal configurations respectively. Notice, that $\text{estimate}_{\text{snake}}\,(x_n, y_n, x_n^+, y_n^+)$ can be calculated as sum of distances between several sections multiplied by number of moves needed to travel a unit of distance in that section. However, as different shapes of snake may occur, this calculation may not be exact. Next, let $\text{cost}_1(x_n, y_n, x_n^+, y_n^+): \Omega_n \times \Omega_n \longrightarrow \mathbb{N}_0$ be a functional that calculates exact number of moves necessary to place given two pebbles individually. As the case of individual pebbles is not distorted by any irregularities (such as different shapes as in the case of snake) the number of moves can be calculated exactly – again it is the sum of distances between given sections multiplied by the number of moves needed to travel unit distance in the individual sections.

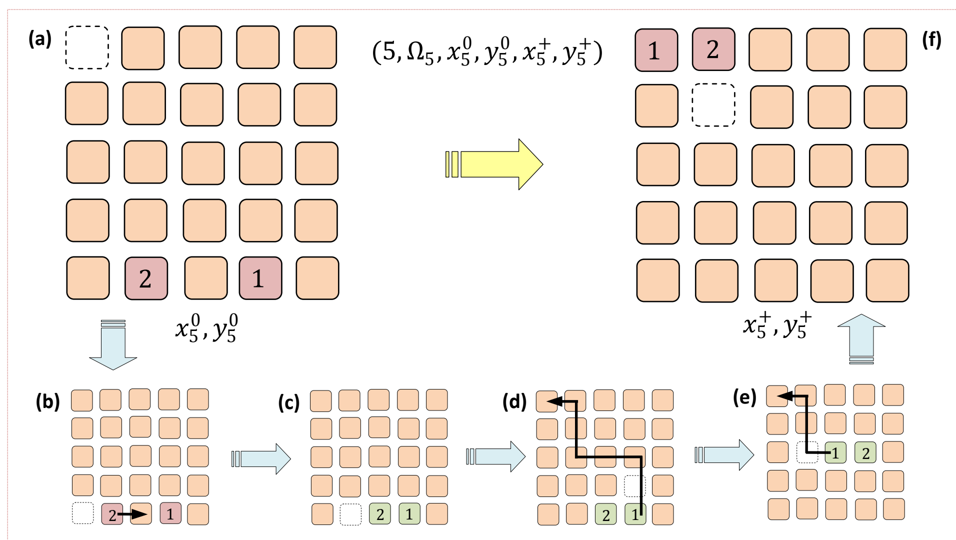


Figure 5. *Illustration of snake formation.* A snake will be formed by moving pebble 2 towards pebble 1 and then the whole snake will move to its goal location. The other way of forming a snake is to move pebble 1 towards pebble 2 and then to move the whole snake.

A preliminary experimental evaluation has shown that it suitable to use the following decision rule: if $\min\{\text{estimate}_{\text{snake}}\,(x_n, y_n, x_n^+, y_n^+)(p,q)$, $\text{estimate}_{\text{snake}}\,(x_n, y_n, x_n^+, y_n^+)\,(p,q)\} < 1.2\text{cost}_1(x_n, y_n, x_n^+, y_n^+)(p,q)$ holds then it is tried to form a snake in the better of two ways and to compare the number of moves when snake is used with $\text{cost}_1(x_n, y_n, x_n^+, y_n^+)(p,q)$. If snake is still better then it is actually used to produce sequence of moves into the solution. Otherwise, the original way of placement of pebbles one by one is used.

The main function *Place-Pebbles* for placing a pair of pebbles using snake like motions is shown using pseudo-code as Algorithm 2. It is supposed that the function is used within the main loop of the solving algorithm (Algorithm 1). Several primitives, which all gets current configuration of pebbles as its first two parameters, are used within Algorithm 2: a function *Move-Vacant* moves the vacant position to a specified new location; a function *Place-Pebble* implements the pebble placement process from the original algorithm of Parberry – here it is used as generic procedure to move pebble from one position to another. Finally, a *Snake-Composed-Movement* is a function that implements composed movements of a specified snake; two positions are specified – the current position of the head of snake and the new position for the head. It is also assumed that movement of the snake does not interfere with already placed pebbles. An example of snake formation and its placement is shown in Figure 5.

4.3. *Discussion on Longer Snakes*

We have also considered usage of snakes of length greater than 2. However, certain difficulties preclude using them effectively. There are many more options how to form a snake of length greater than 2. In the case of length k, there are at least $k!$ basic options how a snake can be formed (the order of pebbles is determined and then the snake collects pebbles in this order). Moreover, those do not include all the options (for example, it may be beneficial to form two snakes instead of a long one and so on). Therefore considering all the options and choosing the best one is computationally infeasible. Hence, using snakes of length 2 seems to be a good trade-off.

4.4. *Theoretical Analysis*

Although our new algorithm produces locally better sequence of moves for placing a pair of pebbles, it may not be necessarily better globally. Consider that different way of placing the pair of pebbles rearranges other pebbles differently as well, which may influence subsequent movements. Hence, theoretical analysis is

quite difficult here. To evaluate the benefit of the new technique in a more realistic manner, we need some experimental evaluation. Nevertheless, theoretical analysis of worst cases can be done at least to get basic insight.

It has been shown that the original algorithm can always find a solution of the length at most $5n^3 + \frac{9}{2}n^2 + \frac{19}{2}n - 89$; that is, $5n^3 + \mathcal{O}(n^2)$ [6].

Proposition 1 (Worst-case Solution Length). Our new algorithm based on snakes can always produce a solution to a given instance of the $(n^2 - 1)$-puzzle of the length of at most $\frac{14}{3}n^3 + \mathcal{O}(n^2)$. ■

Proof. It can be observed that the worst situation for the algorithm using snakes is when the two pebbles – let us denote them p and q – that are about to be placed are located in the last row or column. In such a case, we need $14n + \mathcal{O}(1)$ moves in the worst case. Without loss of generality let us suppose both pebbles p and q to be placed in the last row while p is in the first column and q is in the last column. Exactly it is needed: at most $2n - 1$ moves to move the vacant position near q; then at most $5(n - 1)$ moves to move q towards p which forms a snake; and finally $7n + \mathcal{O}(1)$ moves to relocate the snake into the first row of the grid.

The algorithm needs to place $n - 1$ pairs of pebbles and one pebble individually. Observe that moving one pebble individually to its goal position requires at most $8n$ moves. Hence, the first row and the first column requires at most $14n^2 + c_1n + c_0$ moves where $c_0, c_1 \in \mathbb{R}$ with $c_0, c_1 \geq 0$. Let $M(n)$ denotes number of moves needed to solve the $(n^2 - 1)$-puzzle of size $n \times n$ then it holds that $M(n) \leq M(n-1) + 14n^2 + c_1n + c_0$. The solution of this inequality is $M(n) = \frac{14}{3}n^3 + \mathcal{O}(n^2)$. ■

Proposition 2 (Worst-case Time Complexity). Our new algorithm based on snakes has the worst case time complexity of $\mathcal{O}(n^3)$. ■

Proof. The total time consumed by calls of *Move-Vacant* and *Place-Pebble* is linear in the number of moves that are performed. The time necessary to find shortest path avoiding already placed vertices is linear as well since the path has always a special shape that is known in advance (diagonal followed by horizontal or vertical). There is no need to use any path-search algorithm.

Time necessary for calculating $\text{estimate}_{\text{snake}}$ is at most the time necessary to finish the call of *Place-Pebble*, that is, linear in the number of moves again.

Finally, we need to observe that the call of *Snake-Composed-Movement* consumes time linear in the number of moves again since first the shortest path of the

special shape needs to be found and then a snake needs to be moved along the path. ■

5. Application of 'Snakes' in Cooperative Path-Finding

Promising theoretical and preliminary experimental results from the application of the idea of 'snakes' in solving $(n^2 - 1)$-puzzle inspired us to extend the idea to a closely related problem of *cooperative path-finding* (CPF). The task in cooperative path-finding is to relocate a set of abstract robots over a given undirected graph in a non-colliding way so that each robot eventually reaches its goal vertex [12]. Similarly as in $(n^2 - 1)$-puzzle robots can move into unoccupied vertex while no other robot is allowed to enter the same target vertex at the same time. The natural requirement in CPF is that at least one vertex is empty in the input CPF instance to allow robots to move. Unlike the situation in $(n^2 - 1)$-puzzle, CPF allows multiple robots to move simultaneously provided there are multiple vacant vertices.

The $(n^2 - 1)$-puzzle is thus clearly a special variant of CPF where there is only one unoccupied vertex in the graph and the graph, where pebbles (robots) move, has a special structure of the 4-connected grid. A possible application of snakes in CPF is further supported by the fact that several polynomial-time rule-based algorithms that address CPF such as *BIBOX* [13], *PUSH-and-SWAP* [4], and *PUSH-and-ROTATE* [23] relocate robots one by one over the graph towards their goal locations. That is, in the same way as it is done in the algorithm of Parberry. Hence, these algorithms are candidates for integrating snake movements into their solving process. In this work, we have chosen the *BIBOX* algorithm that has been first introduced in [13] for such a modification.

5.1. *Cooperative Path Finding Formally*

Cooperative path-finding takes place over an undirected graph $G = (V, E)$ where $V = \{v_1, v_2, \dots, v_n\}$ is a finite set of vertices and $E \subseteq \binom{V}{2}$ is a set of edges. The configuration of robots over the graph is modeled by assigning them vertices of the graph. Let $R = \{r_1, r_2, \dots, r_\mu\}$ be a finite set of *robots*. Then, a configuration of robots in vertices of graph G will be fully described by a *location* function $\alpha: R \longrightarrow V$; the interpretation is that an robot $r \in R$ is located in a vertex $\alpha(r)$. At most one robot can be located in a vertex; that is α is a uniquely invertible function. A generalized inverse of α denoted as $\alpha^{-1}: V \longrightarrow R \cup \{\bot\}$ will provide us a robot located in a given vertex or $\bot$ if the vertex is empty.

Definition 7 (***Cooperative Path-Finding***). An instance of *cooperative path-finding* problem (CPF) is a quadruple $\Sigma = [G = (V, E), R, \alpha_0, \alpha_+]$ where location functions α_0 and α_+ define the initial and the goal configurations of a set of robots R in G respectively. □

The dynamicity of the model assumes a discrete time divided into time steps. A configuration α_i at the i-th time step can be transformed by a transition action which instantaneously moves robots in the non-colliding way to form a new configuration α_{i+1}. The resulting configuration α_{i+1} must satisfy the following *validity conditions*: (4)

(i) $\forall r \in R$ either $\alpha_i(r) = \alpha_{i+1}(r)$ or $\{\alpha_i(r), \alpha_{i+1}(r)\} \in E$ holds (robots move along edges or not move at all),

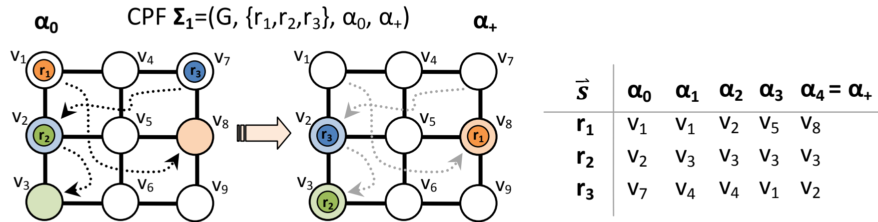
(ii) $\forall r \in R \;\; \alpha_i(r) \neq \alpha_{i+1}(r) \Rightarrow \alpha_i^{-1}(\alpha_{i+1}(r)) = \perp$ (5)
(robots move to vacant vertices only), and

(iii) $\forall r, s \in A \;\; r \neq s \Rightarrow \alpha_{i+1}(r) \neq \alpha_{i+1}(s)$ (6)
(no two robots enter the same target/unique invertibility of the resulting configuration).

The task in cooperative path finding is to transform α_0 using above valid transitions to α_+. An illustration of CPF and its solution is depicted in Figure 6.

Definition 8 ***(Solution).*** A *solution* of a *makespan* m to a cooperative path finding instance $\Sigma = [G, R, \alpha_0, \alpha_+]$ is a sequence of configurations $\vec{s} = [\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_m]$ where $\alpha_m = \alpha_+$ and α_{i+1} is a result of valid transformation of α_i for every $= 1, 2, \ldots, m - 1$. □

The number $|\vec{s}| = m$ is a *makespan* of solution $\vec{s}$. It is known that deciding whether there exists a solution of CPF of a given makespan is NP-complete [8, 15].



$\vec{s}$	α_0	α_1	α_2	α_3	$\alpha_4 = \alpha_+$
r_1	v_1	v_1	v_2	v_5	v_8
r_2	v_2	v_3	v_3	v_3	v_3
r_3	v_7	v_4	v_4	v_1	v_2

Figure 6. *An example of cooperative path-finding problem (CPF).* Three robots r_1, r_2, and r_3 need to relocate from their initial positions represented by α_0 to goal positions represented by α_+. A solution of makespan 4 is shown.

5.2. ***Introducing 'Snakes' into the BIBOX Algorithm***

We will briefly recall basics of the *BIBOX* algorithm before the improvement with snakes will be integrated into it. The comprehensive description and evaluation of the algorithm is given in [16] to which we refer the reader for further details. The algorithm is designed for CPFs over *biconnected graphs* with at least two unoccupied vertices (modifications for single unoccupied vertex exist as well [14]).

Definition 9 ***(connected graph).*** An undirected graph $G = (V, E)$ is *connected* if $|V| \geq 2$ and for any two vertices $u, v \in V$ such that $u \neq v$ there is an undirected path connecting u and v. □

Definition 10 **(*biconnected graph, non-trivial).*** An undirected graph $G = (V, E)$ is *biconnected* if $|V| \geq 3$ and the graph $G' = (V', E')$, where $V^{'} = V \setminus \{v\}$ and $E^{'} = \{\{u, w\} | u, w \in V \wedge u \neq v \wedge w \neq v\}$, is connected for every $v \in V$. A biconnected graph not isomorphic to a cycle will be called *non-trivial* biconnected graph. □

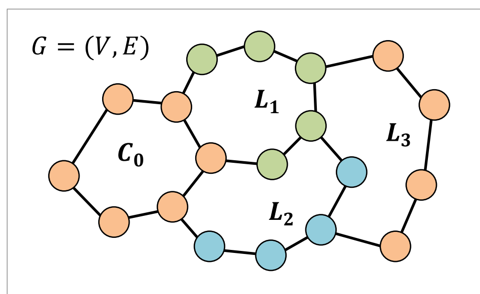


Figure 7. *Example of* ***biconnected graph***. An ear decomposition is illustrated. The graph can be constructed by starting with cycle C_0 and by gradually adding ears L_1, L_2, and L_3.

Observe that, if a graph is biconnected, then every two distinct vertices are connected by at least two *vertex disjoint paths* (equivalently, there is a cycle containing both vertices; only internal vertices of paths are considered when speaking about vertex disjoint paths - vertex disjoint paths can intersect in their *start points* and *endpoints*). An example of biconnected graph is shown in Figure 7.

An algorithmically important property of biconnected graphs is that every biconnected graph can be constructed from a cycle by adding sequence of ears to the currently constructed graph [18, 21, 22]. The *BIBOX* algorithm is substantially based on this property. Consider a graph $G = (V, E)$; the new ear with respect to G is a sequence $L = [u, w_1, w_2, \dots, w_h, v]$ where $h \in \mathbb{N}_0$, $u, v \in V$ such that $u \neq v$ (called *connection vertices*) and $w_i \notin V$ for $i = 1,2, \dots, h$ (w_i are fresh vertices). The result of the addition of the ear L to the graph G is a new graph $G^{'} = (V^{'}, E^{'})$ where $V^{'} = V \cup \{w_1, w_2, \dots, w_h\}$ and either $E^{'} = E \cup \{\{u, v\}\}$ in the case of $h = 0$ or $E^{'} = E \cup \{\{u, w_1\}, \{w_1, w_2\}, \dots, \{w_{h-1}, w_h\}, \{w_h, v\}\}$ in the case of $h > 0$. Let

the sequence of ears together with the initial cycle be called an *ear decomposition* of the given biconnected graph. Again, see Figure 7 for illustrative example.

Lemma 1 ***(ear decomposition)*** [18, 21, 22]**.** Any biconnected $G = (V, E)$ graph can be obtained from a cycle by a sequence of operations of adding an ear. ■

The important property of the construction of a biconnected graph according to its ear decomposition is that the currently constructed graph is biconnected at every stage of the construction. The algorithm for solving CPFs over biconnected graphs can proceed inductively according to the ear decomposition by arranging robots into individual ears – after finishing placement of robots into an ear, the problem reduces to a problem of the same type but on a smaller graph without the currently solved ear.

As the *BIBOX* algorithm has been already thoroughly published, its enhancement with snakes described using pseudo-code has been deferred to Appendix B (Algorithm 3). The idea behind using snakes within the *BIBOX* algorithm is similar as in the case of the algorithm of Parberry.

Again, snakes of length 2 are used within the modified *BIBOX* algorithm. Consider that robots r and s are two consecutive robots within the processed ear L_i. In the original algorithm, they are moved one by one towards the ear connection vertex and stacked inside the ear by its rotation afterwards; that is, relocation and stacking inside the ear of r and s is done separately. When snake reasoning is used, it is first checked if r and s are close enough to each other before r is relocated towards the ear connection vertex. If it is the case, then r and s are relocated together in tandem until r reaches the connection vertex. After such relocation, robot s is next to the connection vertex and can be then stacked into the ear quickly. If robots r and s are too far from each other, then the original relocation of both robots separately is used. The process of relocation of two consecutive robots is implemented by procedure *Move-Robot-Snake* within the pseudo-code of Algorithm 3.

Let us now clarify what does it mean that robots are close enough to each other and what relocation in tandem means. When considering if snake based relocation pays-off, a simple distance heuristic is used. The cost of relocation is estimated by the length of shortest path between the original and target location. Let v be an ear connection vertex, α current configuration of robots and let $\text{dist}_G(u, v)$ for $u, v \in G$ denote the shortest path between u and v in $G = (V, E)$. Then robots r and s are relocated in tandem if:

$$\text{dist}_G(v, \alpha(r)) + \text{dist}_G(\alpha(r), \alpha(s)) < \text{dist}_G(v, \alpha(r)) + \text{dist}_G(v, \alpha(s)) \quad (7)$$

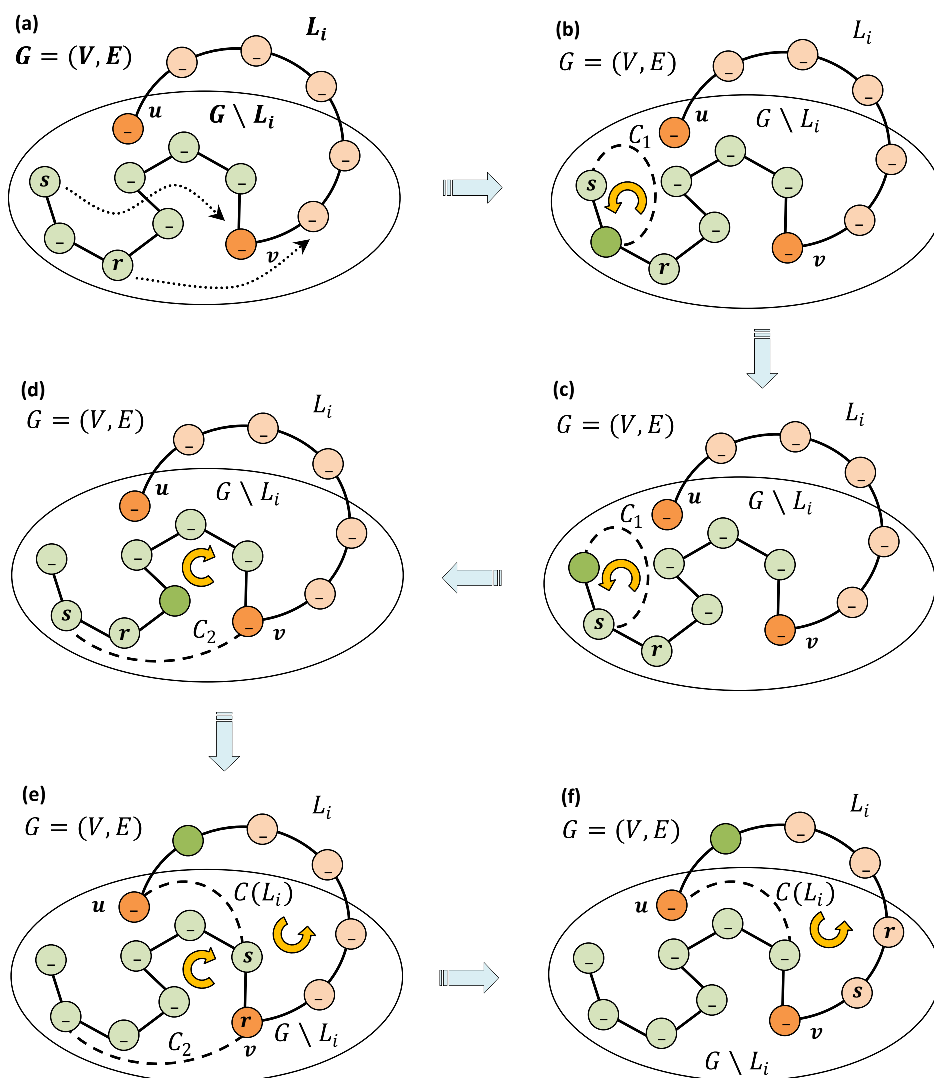


Figure 8. *Illustration of using* ***snakes of size 2*** *within the* ***BIBOX*** *algorithm.* A pair of robots r and s needs to be stacked into ear L_i next to each other. They first need to be moved towards ear connection vertex v. Shortest paths connecting robot locations with vertex v are depicted. According to distance heuristic it is decided that r and s should be relocated together in tandem. Thus, robot s is moved next to r by rotating cycle C_1 (stages (b), (c)). Then, r and s moves like a snake in tandem by rotating cycle r and C_2 until r appears in the ear connection vertex v (stage (d)). Finally, robot r is stacked into ear L_i by rotation of cycle $C(L_i)$ associated with the ear (stage (e)). As robot s has been next to r all the time, it consequently moved to its target vertex during the last rotation as well (stage (f)). The symbol _ stands for an anonymous robot.

That is, if relocation of s towards r and relocation of r and s in tandem towards v seems to be less costly than relocation of r and s towards v separately.

The original relocation of a robot r within the *BIBOX* algorithm is done by finding a cycle which includes the target vertex and robot r. One vertex within the cycle is made unoccupied which enable rotation of the cycle. Robot r is moved towards its target by rotating the cycle until r appears in the target vertex. The original relocation is implemented by *Move-Robot* procedure in the pseudo-code.

The tandem relocation of a pair of robots uses the very same idea. First, robot s is moved next to r by the original way of relocating robots (*Move-Robot*). Then a cycle containing the edge whose endpoints are occupied by r and s respectively. The cycle rotated until r reached its target. Throughout the series of rotations of the cycle, robots r and s are preserved to stay next to each other, which eventually means that s is close to its target at the end of tandem relocation. The tandem relocation is implemented by *Move-Robot-Snake* within the pseudo-code. The illustration of the process of tandem relocation of a pair of robots is shown in Figure 8.

6. Experimental Evaluation

An experimental evaluation is necessary to explore qualities of our new snake-based improvements to the algorithm of Parberry and to the *BIBOX* algorithm.

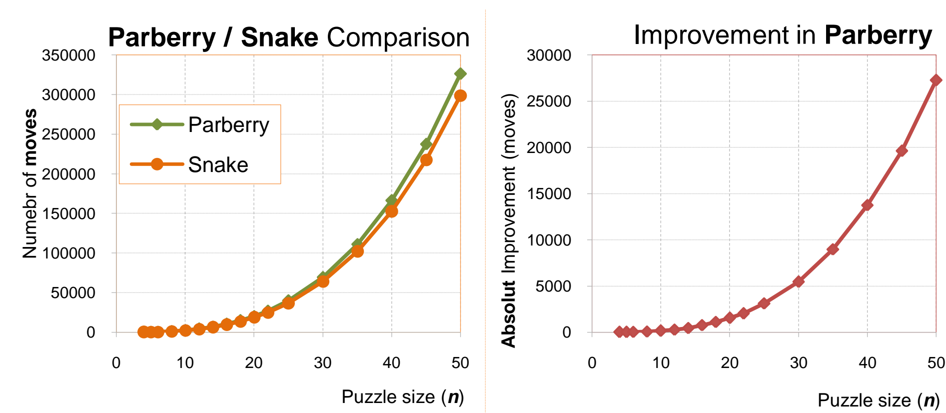


Figure 9. *Comparison of the original* ***Parberry's algorithm*** *and its* ***snake-based improvement*** *in terms of total number of steps.* Comparison has been done for several sizes of the puzzle ranging from 3 to 50. Forty random instances were generated for each size of the puzzle. The average number of moves for both algorithms is shown in the left part. The absolute improvement that can be achieved by using snakes is shown in the right part.

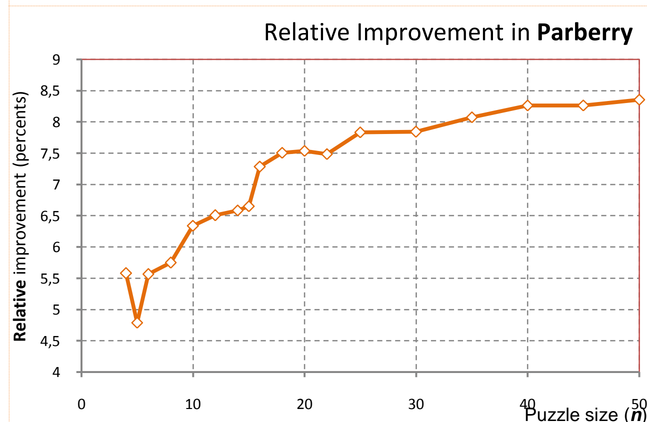
In case of snake-based improvement to Parberry's algorithm, we have only the upper bound estimation of the total number of steps so far which however does not show that the new algorithm actually produces fewer moves. Thus, a thorough empiric evaluation needs to be done.

The snake-based improvements to Parberry's algorithm as well as the original algorithm were implemented in C++ to make experimental evaluation possible. The snake-based reasoning was also integrated into the *BIBOX* algorithm which original C++ implementation was available [13] so only minor changes to the code needed to be made. A series of tests has been conducted to measure the total number of moves performed by each algorithm. The runtime necessary to solve the given instance has been measured too.

Relative Improvement in Parberry's Algorithm	
n	Length Improvement (%)
4	5.58
5	4.79
6	5.57
8	5.75
10	6.34
12	6.51
14	6.59
16	6.66
18	7.29
20	7.51
22	7.54
25	7.49
30	7.84
35	7.84
40	8.07
45	8.26
50	8.26

Table 1. ***Relative improvement*** *achieved by using snakes with respect to the original algorithm.* Again, the improvement has been measured for several sizes of the puzzle ranging from 4 to 50. For each size, 40 random instances were generated and the average improvement was calculated.

Figure 10. *Illustration of the* ***trend*** *in the average improvement.* It can be observed that the relative improvement tends to stabilize between 8% and 9% as instances are getting larger.



Regarding the choice of testing puzzles, we followed the benchmark generation proposed by Korf and Taylor in [2] where random instances of the (5^2-1)-puzzle were used. There is an experimental evidence that solving random instances of the (5^2-1)-puzzle optimally is difficult. Our experimental evaluation has been done for random puzzles of sizes of ranging from 4 to 50 (that is, n was ranging from 4 to 50). For each size of the puzzle, 40 solvable instances with random initial and goal configuration of pebbles were generated (notice, that sol-

vability can be detected by permutation parity check). Each generated instance was then solved by all the tested algorithms, that is, by Parberry's algorithm, its snake-based improvement, by the *BIBOX* algorithm, and *BIBOX* with snake improvement.

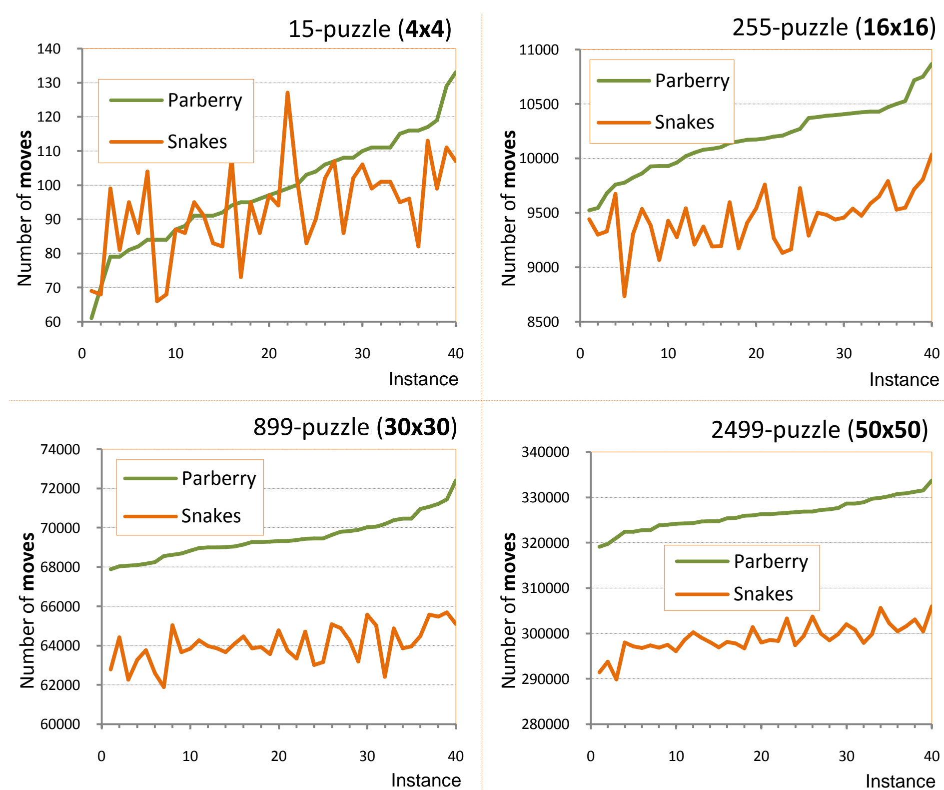


Figure 11. *Development of the improvement in* ***Parberry's algorithm*** *with the growing size of the puzzle instance.* Comparison of the number of moves conducted by the algorithm of Parberry and by its snake-based improvement is shown for four puzzles of the increasing size. Individual instances for each size of the puzzle are sorted according to the increasing number of steps made by Parberry's algorithm. It is observable that a worsening after applying snake-based approach may appear in small instances. Nevertheless, the improvement is becoming stable (between 8-9%) for larger instances.

A series of tests with CPFs over biconnected graphs with various ear decompositions has been also done to evaluate benefits of snakes in situations structurally different from those in (n^2-1)-puzzle. Note that (n^2-1)-puzzle takes place over 4-connected grid, which is a very special case of biconnected graph. Moreover ear decompositions used in the *BIBOX* algorithm on puzzles has most of ears

of size 1, which is a yet more special case. Hence, evaluation of snake-based improvements in the *BIBOX* algorithm on biconnected graphs with ear decompositions consisting of longer ears has been done. Tests with CPFs were also focused on evaluation of snake-based improvements with various numbers of robots in the instance (multiple vacant vertices may be available).

The complete source code and raw experimental data are provided at the website: http://ktiml.mff.cuni.cz/~surynek/research/j-puzzle-2014 to allow full reproducibility of presented results and own experiments with snake-based improvements in tested algorithms.

6.1. *Competitive Comparison of Parberry's Algorithm with Snakes*

The competitive comparison of the total number of moves made by the algorithm of Parberry and its snake-based improvement is shown in Figure 9. The improvement achieved by snake-based approach is illustrated as well. For each size of the instance, average out of 40 random instances is shown.

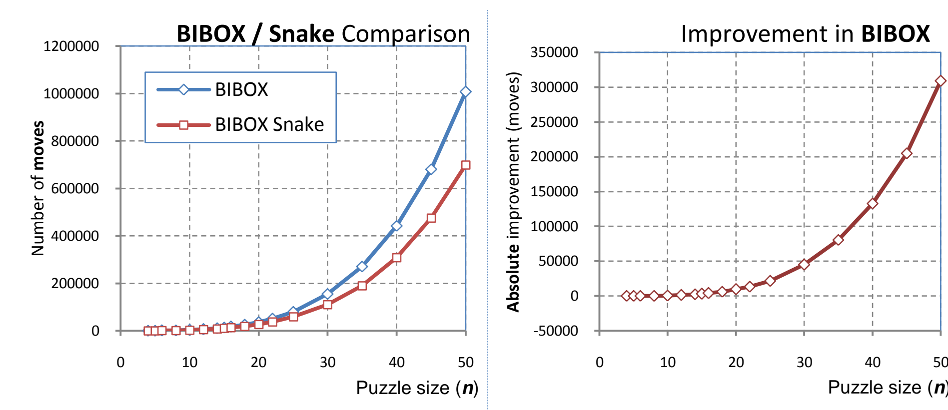


Figure 12. *Comparison of the original* ***BIBOX algorithm*** *and its modification that uses* ***snakes***. The experimental setup is the same as in the case of Parberry's algorithm – $(4^2 - 1)$ to $(50^2 - 1)$ puzzles with 40 random instances for each size were used for evaluation. The average number of steps over 40 instances is shown for each size of the puzzle. The *BIBOX* algorithm needs approximately 3-times more steps than Parberry's algorithm to solve an instance of the $(n^2 - 1)$-puzzle. The absolute improvement in terms of the number of steps after introducing snake-based movements into *BIBOX* algorithm is larger in absolute number of steps than after using snakes within the Parberry algorithm.

It is observable that the growth of the number of moves for growing size of the instance is polynomial. Next, it can be observed that snakes achieve a stable improvement, which is proportional to the total number of moves. The more detailed

insight into the achieved improvement of the total number of moves is provided by Figure 10 and Table 1. It clearly indicates that the improvement is becoming stable between 8% and 9% with respect to the original algorithm, as instances are getting larger.

6.2. *Parberry's Algorithm on Individual Puzzle Instances*

Comparison of the total number of moves on the individual instances of various sizes is shown in Figure 11. These results show that using snakes, even though it is locally a better choice, can lead to global worsening of the solution. This phenomenon sometimes occurs exclusively on small instances. Here it is visible for instances of the size of 4×4.

On larger instances, the local benefit of using snakes dominates over any local worsening of the configuration so there is stably significant improvement of 7% and 9%. Notice, that this is not the average improvement calculated from several instances; this is improvement on a single instance.

6.3. *Competitive Comparison of BIBOX Algorithm with Snake Improvement*

The *BIBOX* algorithm can be used to solve $(n^2 - 1)$-puzzle instances, as they are special cases of CPF. Note however, that the general algorithm for biconnected graphs needs at least two unoccupied vertices. These two unoccupied vertices are needed to arrange robots/pebbles in the initial cycle of the ear decomposition while just one vacant vertex is sufficient to arrange pebbles in regular ears [16]. As we are working with graphs of fixed 4-connected structure in $(n^2 - 1)$-puzzle a slight adaptation of the *BIBOX* algorithm that arranges pebbles in the initial cycle by fixed rules using just one vacant vertex is possible. Throughout the solving process, an ear decomposition of the 4-connected grid where internal ears consist of single internal vertex was used.

Results from the comparison of the number of steps in solutions of the puzzle generated by the *BIBOX* algorithm and its snake-based improvement are shown in Figure 12. Observe that the *BIBOX* algorithm generates approximately 3 times larger solutions than the algorithm of Parberry. This is however natural result as *BIBOX* algorithm is more general for biconnected graphs and does not exploit the advantage of a priori knowledge that the underlying graph is a 4-connected grid. It is also noticeable that using snakes in case of the *BIBOX* algorithm saves much more steps in absolute terms than in the case of Parberry's algorithm.

The relative improvement after introducing snakes into the *BIBOX* algorithm as shown in Figure 13 and Table 2 is around 30% in larger puzzle instance. In

small instances even worsening may appear which is caused by inaccuracy of the distance heuristic (7) which does not take into account that the number of performed moves does not need to correspond to shortest paths and that the second robot (denoted as s in section 5.2) may relocate after relocating the first robot (denoted as r).

Relative Improvement in BIBOX Algorithm	
n	**Length Improvement** (%)
4	-9.29
5	-1.46
6	2.72
8	13.98
10	18.48
12	19.71
14	22.08
16	22.42
18	23.54
20	24.45
22	26.03
25	26.46
30	26.88
35	29.04
40	29.79
45	30.07
50	30.14

Table 2. ***Relative improvement*** *achieved by using snakes in the BIBOX algorithm.* Again, the improvement has been measured for several sizes of the puzzle ranging from $(4^2 - 1)$ to $(50^2 - 1)$ with 40 random instances per size. Relative improvement is significantly larger in the case of *BIBOX* algorithm than in Parberry's algorithm.

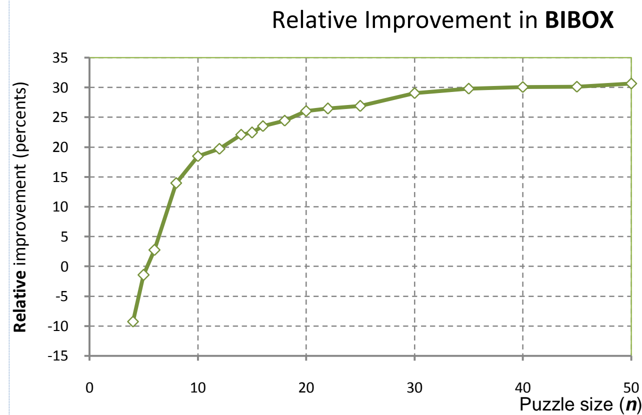


Figure 13. *Illustration of the* ***trend*** *in the average improvement in the BIBOX algorithm.* It can be observed that the relative improvement tends to stabilize around 30% as instances of the $(n^2 - 1)$-puzzle are getting larger.

6.4. *BIBOX Algorithm on Individual Puzzle Instances*

Similarly as in the case of Parberry's algorithm, we show results of test of the *BIBOX* algorithm over individual instances of the puzzle. Results are shown in Figure 14. In small instances, relatively significant worsening may appear after using snakes in algorithm *BIBOX*. On the other hand, in large instances significant improvement over 30% can be achieved. Again the worsening in small instances can be explained by inaccuracy of distance heuristic (7) as in small instances stronger interference between two relocated robots is more likely.

6.5. *Evaluation of Using Snakes in BIBOX Algorithm on CPFs*

Promising results obtained in solving (n^2-1)-puzzle by snake-improved algorithms inspired us to evaluate snake-based version of the *BIBOX* algorithm on instances of CPF over biconnected graphs that are structurally different from the puzzle.

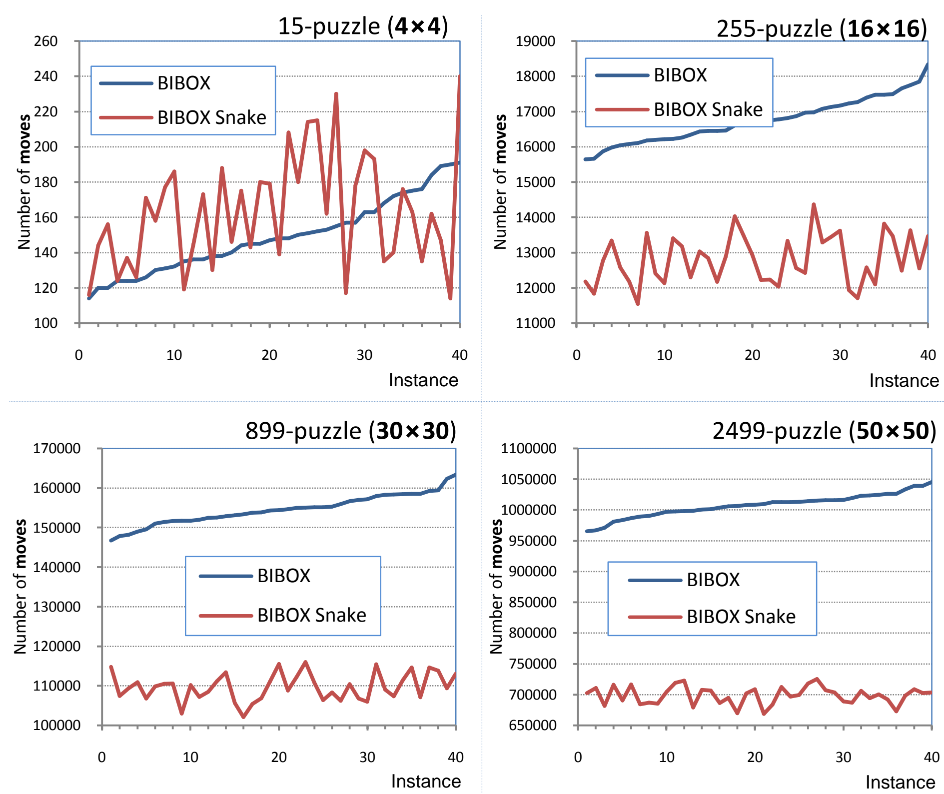


Figure 14. *Development of the improvement in the* ***BIBOX algorithm*** *with the growing size of the* ***puzzle*** *instance.* The improvement is shown for all the 40 random instances for several sizes of the (n^2-1)-puzzle. Worsening may appear after using snakes in the *BIBOX* algorithm in small instances – the same behavior can be observed in Parberry's algorithm. Nevertheless, the improvement is becoming stable around 30% in larger instances.

Note that the ear decomposition of the 4-connected grid, where (n^2-1)-puzzle takes place, is quite special – it consists of ears having just one internal vertex. Hence, it would be interesting how snake improvement behaves in *BIBOX* algorithm over biconnected graphs with longer ears.

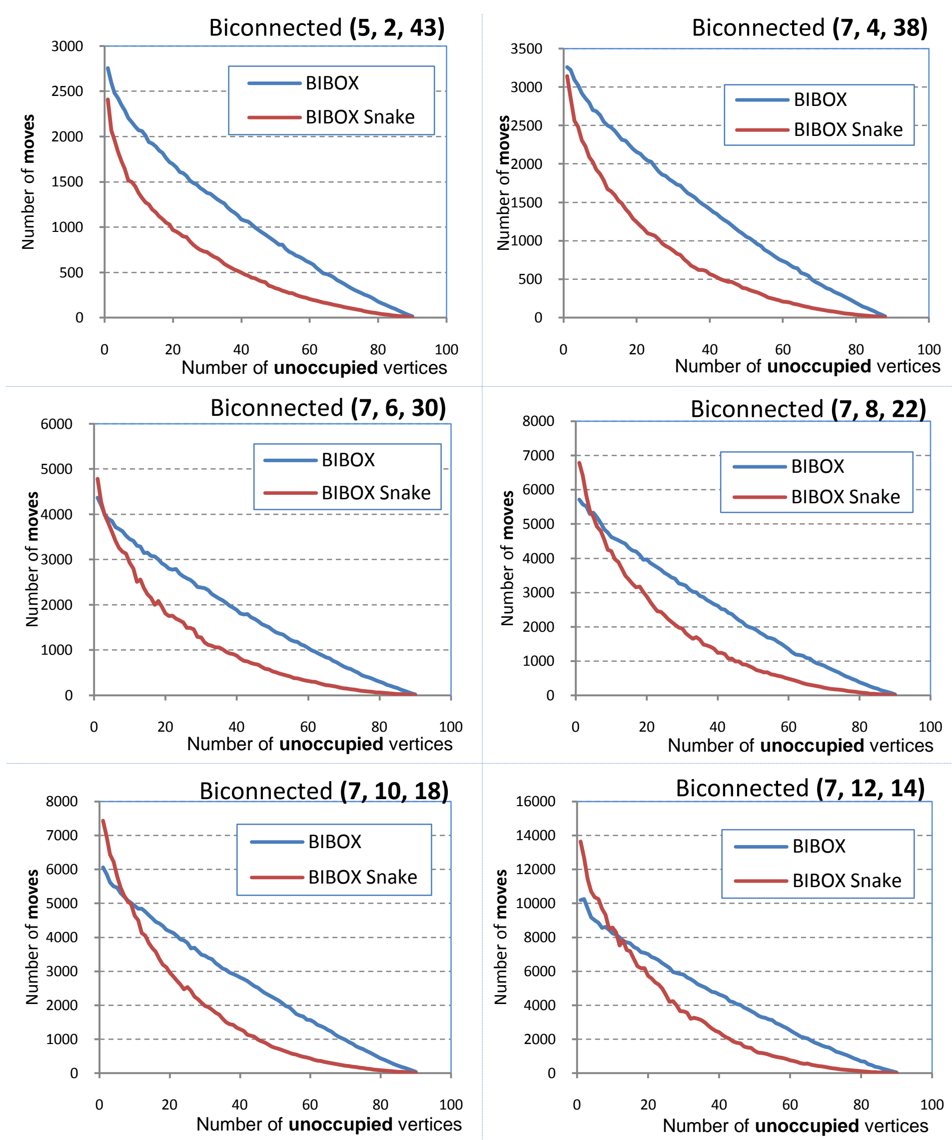


Figure 15. *Improvements after* ***introducing snakes into BIBOX*** *algorithm on* ***biconnected*** *graphs.* Biconnected graph with different sizes of ears in ear decompositions were tested – $Biconnected(c, e, m)$ stands for a biconnected graph with initial cycle of size c, average internal ear size e, and number of ears m. Also, the number of robot changed from almost fully occupied graph (one vacant vertex) to a graph occupied by single robot. Again, the average number of steps in solutions to 40 random instances for each number of occupied vertices is shown. In almost all the cases using snakes brings significant improvement of the total number of steps. There is also observable tendency that snakes appear more beneficial if multiple vacant vertices are available. The only case where snakes cause worsening consist of almost fully occupied graphs with relatively long ears in the ear decomposition.

We also evaluated snake-based improvements over CPF instances with various numbers of robots; that is, when multiple vacant vertices are available which may affect snake formation and interference between two relocated robots significantly. We expect higher accuracy of the distance heuristic (7) in cases of with fewer robots (more vacant vertices) as there should be weaker interference between two relocated robots.

Several random biconnected graphs were constructed over which random CPF instances were generated. Random instances over fixed graph are obtained by generating random initial and goal configuration of robots. Results from the evaluation on CPFs over biconnected graphs are shown in Figure 15.

The notation $Biconnected(c, e, m)$ denotes random biconnected graph with an ear decomposition where the initial cycle consists of c vertices, the average number of internal vertices of ears is e, and the total number of ears is m. Several biconnected graphs containing approximately 90 vertices were used in the evaluation.

Results indicate that snake-based improvement is more efficient if there are more unoccupied vertices in the instance, which conforms to our expectation. Up to 50% moves can be saved after employing snakes in CPF solving over biconnected graphs with approximately half of vertices occupied by robots and even larger proportion (up to 80%) of moves can be saved in sparsely occupied graphs.

Worsening after using snakes appears more frequently than in the case of puzzle. It may appear especially with long ears in densely occupied graphs. This behavior can be explained by the fact that there are fewer alternative paths in biconnected graphs with long ears and by the fact that robot can sometimes be influenced by other robots in densely occupied graph which can divert it from its direction. Due to absence of alternative paths the diversion cannot be repaired as easily as in the case of (n^2-1)-puzzle – in other words distance heuristic (7) is quite inaccurate in such cases.

6.6. ***Runtime Measurement***

Finally, results regarding runtime are presented in Table 3. The average runtime for puzzles of size up to 50×50 are shown. Expectably, snake-based improvement of Parberry's algorithm is slower as it makes decisions that are more complex (in fact, it is running the original algorithm plus snake placement to compare if snake is locally better). Nevertheless, the slowdown is well acceptable.

Notice that both algorithms – Parberry's and its snake-based improvement – are capable of solving puzzles with solutions consisting of hundreds of thousands

of moves almost immediately. Hence, it can be concluded that both algorithms scales up extremely well and they can be used in on-line applications.

Table 3. ***Runtime***[1] ***measurements*** *of the algorithm of Parberry, BIBOX algorithm and their snake variants.* Average time is calculated for each size of the puzzle out of 40 runs with different random setups. It can be observed that both algorithms scale up well.

	n	10	20	30	40	50
Time (seconds)	**Parberry**	< 0.10	< 0.10	< 0.10	0.10	0.10
	Parberry/Snakes	< 0.10	< 0.10	< 0.10	0.10	0.19
	BIBOX	< 0.10	3.52	37.39	211.95	835.06
	BIBOX/Snakes	< 0.10	2.99	30.25	168.50	657.57

The absolute time in the case of *BIBOX* algorithm is much worse since the algorithm works on general biconnected graph while Parberry's algorithm works on fixed grid thus there is much less decisions in Parberry's algorithm. We also would like to mention that we used the original implementation of algorithm *BIBOX* provided by the author [13] in above measurements. We expect that the implementation can be further optimized. The important result is the difference between the original and snake-improved version.

It is noticeable that although snakes require more complex computations these in fact should not increase the runtime significantly – the distance heuristic (7); that is, the distance between currently placed robot and the next robot to be placed can be calculated by looking into table containing all-pairs of shortest paths. The time needed for this pre-calculation is dominated by the runtime of the rest of the *BIBOX* algorithm theoretically as well as empirically. The performance in terms of the runtime is better when snakes are utilized because the algorithm does need to produce significantly fewer moves.

6.7. *Summary of Experimental Evaluation*

The conducted experimental evaluation clearly shows the hat snake-based reasoning integrated to the original algorithm of Parberry as well as to the *BIBOX* algorithm brings significant improvements in terms of the quality of generated solutions (defined as total number of moves). This claim is experimentally supported in both (n^2-1)-puzzle and yet more distinctively in cooperative path finding instances on biconnected graphs.

[1] All the tests with Parberry's algorithm were run on a commodity PC with CPU Intel Core2 Duo 3.00 GHz and 2 GB of RAM under Windows XP 32-bit edition. The C++ code was compiled with Microsoft Visual Studio 2008 C++ compiler. Tests with the *BIBOX* algorithm were run on an experimental server with the 4-core CPU Xeon 2.0GHz and 12GB RAM under Linux kernel 3.5.0-48.

Experiments support the claim that using snakes greedily, that is, if they are locally better, leads to global improvement of solution even though the current configuration may be worsened sometimes from the global point of view. As instances are getting larger, the improvement tends to stabilize itself between 8% and 9% in average in case of Parberry's algorithm and around 30% in the case of *BIBOX* algorithm on $(n^2 - 1)$-puzzle. On larger instances – that is larger than 30×30 – possible fluctuations towards worsening the solution are eliminated, hence using snakes expectably leads to mentioned improvement on an individual instance.

Runtime measurements show that original Parberry's algorithm and its snake-based improvement solve instances of tested sizes in less than $0.2s$. Thus, it can be concluded that scalability is extremely good.

Surprisingly, snake-based reasoning improves solutions quite dramatically in CPFs on biconnected graphs with longer ears in the decomposition and fewer robots in the graph. In such cases, snakes help the *BIBOX* algorithm to reduce the size of the solution by up to 50% or even 80% in sparsely populated instances.

7. Conclusions and Future Work

We have presented an improvement to the polynomial-time algorithm for solving the $(n^2 - 1)$-puzzle in an on-line mode sub-optimally. The improvement is based on an idea to move pebbles jointly in groups called *snakes*, which was supposed to reduce the total number of moves. The experimental evaluation eventually confirmed this claim and showed that the new algorithm outperforms the original algorithm of Parberry [6] by 8% to 9% in terms of the average length of the solution. Theoretical upper bounds on the worst-case length of the solution are also better for the new algorithm as we have shown. Regarding the runtime, the new algorithm is marginally slower due to its more complex computations, however this is acceptable for any real-life application as the runtime is linear in the number of produced moves (approximately 10^6 moves can be produced per second).

Promising results with snake-based pebble moving in $(n^2 - 1)$-puzzle led us to the idea to try to integrate snake-based movement into methods for solving the problem of *cooperative path-finding* (CPF) of which the $(n^2 - 1)$-puzzle is a special case. We have chosen the *BIBOX* algorithm [16] for integrating snakes as it processes robots in CPFs in a similar way how Parberry's algorithm processes pebbles (that is, one by one and after the robot is placed it does not move any more). Improvements gained after integrating snake-based reasoning into *BIBOX* algorithm were even more significant than in case of Parberry's algorithm. Up to 30% improvement was reached in solving $(n^2 - 1)$-puzzle with algorithm *BI-*

BOX and up to 50% improvement has been reached in CPFs over biconnected graphs with long ears and multiple unoccupied vertices. Moreover, the improvement in CPFs on biconnected graphs has the growing tendency as the number of unoccupied vertices increases.

It will be interesting for future work to add more measures for reducing the total number of moves towards the optimum. Observe that choosing a more promising local rearrangement among several options can be easily parallelized.

We are also interested in generalized variants of the (n^2-1)-puzzle where there is more than one vacant position. These variants are known as the (n^2-k)-puzzle with $k>1$ [14]. Although it seems that obtaining optimal solutions remains hard in this case, multiple vacant positions can be used to rearrange pebbles more efficiently in the sub-optimal approach.

It seems that adapting the *BIBOX* algorithm for snakes of length more than 2 is also possible. A robot can collect the snake along its relocation towards the ear connection vertex. Another open question is how the snake-based approach could perform in the *directed version* of CPF [1, 25]. Unidirectional environment puts additional constraints on relocation and hence solution reduction using snakes may have greater effect.

Finally, it is interesting for us to study techniques for optimal solving of this and related problems; especially the case with small unoccupied space (that is, with $k \ll n^2$). This is quite open area as today's optimal solving techniques [17] can manage only small number of pebbles compared to the size of the unoccupied space.

References

1. **Botea**, A., **Surynek**, P.: *Multi-Robot Path Finding on Biconnected Directed Graphs.* Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015), Austin, TX, USA, AAAI Press, 2015.

2. **Korf**, R. E., **Taylor**, L. A.: *Finding Optimal Solutions to the 24-Puzzle.* Proceedings of the 13th National Conference on Artificial Intelligence (AAAI 1996), pp. 1202-1207, AAAI Press, 1996.

3. **Kornhauser**, D., **Miller**, G. L., and **Spirakis**, P. G.: *Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications.* Proceedings of the 25th Annual Symposium on Foundations of Computer Science (FOCS 1984), pp. 241-250, IEEE Press, 1984.

4. **Luna**, R., **Bekris**, K. E.: *Push and Swap: Fast Cooperative Path-Finding with Completeness Guarantees.* Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011), pp. 294-300, IJCAI/AAAI, 2011.

5. **Michalík**, P.: *Sub-optimal Algorithms for Solving Sliding Puzzles.* Master thesis, Charles University in Prague, Czech Republic, 2011.

6. **Parberry**, I.: *A real-time algorithm for the (n^2-1)-puzzle*. Information Processing Letters, Volume 56 (1), pp. 23-28, Elsevier, 1995.

7. **Ratner**, D. and **Warmuth**, M. K.: *Finding a Shortest Solution for the N×N Extension of the 15-PUZZLE Is Intractable.* Proceedings of the 5th National Conference on Artificial Intelligence (AAAI 1986), pp. 168-172, Morgan Kaufmann Publishers, 1986.

8. **Ratner**, D. and **Warmuth**, M. K.: *NxN Puzzle and Related Relocation Problems.* Journal of Symbolic Computation, Volume 10 (2), pp. 111-138, Elsevier, 1990.

9. **Russel**, S., **Norvig**, P.: *Artificial Intelligence – A modern approach.* Prentice Hall, 2003.

10. **Ryan**, M. R. K.: *Graph Decomposition for Efficient Multi-Robot Path Planning.* Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 2003-2008, IJCAI, 2007.

11. **Ryan**, M. R. K.: *Exploiting Subgraph Structure in Multi-Robot Path Planning.* Journal of Artificial Intelligence Research (JAIR), Volume 31, 2008, pp. 497-542, AAAI Press, 2008.

12. **Silver**, D.: *Cooperative Pathfinding.* Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2005), pp. 117-122, AAAI Press, 2005.

13. **Surynek**, P.: *A Novel Approach to Path Planning for Multiple Robots in Biconnected Graphs.* Proceedings of the 2009 IEEE International Conference on Robotics and Automation (ICRA 2009), pp. 3613-3619, IEEE Press, 2009.

14. **Surynek**, P.: *An Application of Pebble Motion on Graphs to Abstract Multi-robot Path Planning.* Proceedings of the 21st International Conference on Tools with Artificial Intelligence (ICTAI 2009), pp. 151-158, IEEE Press, 2009.

15. **Surynek**, P.: *An Optimization Variant of Multi-Robot Path Planning is Intractable.* Pro-ceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010), pp. 1261-1263, AAAI Press, 2010.

16. **Surynek**, P.: *Solving Abstract Cooperative Path-Finding in Densely Populated Environments.* Computational Intelligence (COIN), Volume 30 (2), pp. 402-450, Wiley, 2014.

17. **Standley**, T.: *Finding Optimal Solutions to Cooperative Pathfinding Problems.* Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI-2010), pp. 173-178, AAAI Press, 2010.

18. **Tarjan**, R. E.: *Depth-First Search and Linear Graph Algorithms.* SIAM Journal on Computing, Volume 1 (2), pp. 146-160, Society for Industrial and Applied Mathematics, 1972.

19. **Wang**, K. C., **Botea**, A.: *Scalable Multi-Robot Pathfinding on Grid Maps with Tractability and Completeness Guarantees.* Proceedings of ECAI 2010 - 19th European Conference on Artificial Intelligence (ECAI 2010), pp. 977-978, Frontiers in Artificial Intelligence and Applications 215, IOS Press, 2010.
20. **Wang**, K. C., **Botea**, A., **Kilby**, P.: *Solution Quality Improvements for Massively Multi-Robot Pathfinding.* Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI 2011), AAAI Press 2011.
21. **West**, D. B.: *Introduction to Graph Theory.* Prentice Hall, 2000.
22. **Westbrook**, J., **Tarjan**, R. E.: *Maintaining bridge-connected and bi-connected components on-line.* Algorithmica, Volume 7, Number 5&6, pp. 433–464, Springer, 1992.
23. **de Wilde**, B., **ter Mors**, A., **Witteveen**, C.: *Push and Rotate: a Complete Multi-robot Pathfinding Algorithm.* Journal of Artificial Intelligence Research (JAIR), Volume 51, pp. 443-492, AAAI Press, 2014.
24. **Wilson**, R. M.: *Graph Puzzles, Homotopy, and the Alternating Group.* Journal of Combinatorial Theory, Ser. B 16, pp. 86-96, Elsevier, 1974.
25. **Wu**, Z., **Grumbach**, S: Feasibility of motion planning on acyclic and strongly connected directed graphs. Discrete Applied Mathematics, Volume 158 (9), pp. 1017 – 1028, Elsevier, 2010.

Appendix A – Analysis of the Average Case

Regarding the average case analysis we will assume in accordance with [6] that every initial configuration of pebbles can occur with the same probability. We will first show that algorithm of Parberry [6] produces $4n^3 - \frac{1}{2}n^2 + \frac{3}{2}n - 70$ moves in the average case. Then we will simulate this analysis also for our snake-based algorithm. Unfortunately, it seems not to be possible to express the average number of moves as any simple formula in the case of the snake-based algorithms. However, we can provide some arguments that the average solution length of the snake-based algorithm is strictly better than that of Parberry's algorithm.

Before we start with proofs of main propositions, we will introduce several technical lemmas. Proofs of these lemmas are omitted since they are easy and rather technical (detailed proofs can be found in [5]).

Lemma 2. The average value of $\mu_n((1,1);(x,y))$ for $x, y \in \{1,2,\dots,n\}$ (that is the average Manhattan distance from the position $(1,1)$) is $n - 1$. ■

Lemma 3. The average value of $\mu_n((1,k);(x,y))$ for $k,x,y \in \{1,2,\dots,n\}$ such that for $x > k$ or for $y > 1$ (that is, for a given k we consider only (x,y) positions that follows the position $(1,k)$ in the top-down/left-right direction) is at most $n - \frac{1}{2}$. ■

The similar result can be obtained for positions in the first column. But here the estimation of the Manhattan distance is lower – namely $n - 1$.

Lemma 4. The number of moves necessary to move a pebble from a position (i,j) to a position $(1,k)$ supposed that the position $(1,k)$ is unoccupied is at most $6\mu_n\big((1,k);(i,j)\big) + 1$. ■

Proposition 3 (Average-case Solution Length - Parberry). The average length of solutions to $(n^2 - 1)$-puzzle produced by Parberry's algorithm is at most $4n^3 - \frac{1}{2}n^2 + \frac{3}{2}n - 70$. ■

Proof. From the Lemma 3 and Lemma 4 we can obtain that the expected number of moves necessary to solve the first row of the puzzle is at most: $n \cdot \big(6 \cdot \big(n - \frac{1}{2}\big) + 1\big) = 6n^2 - 2n$. Similarly for the first column: $(n-1) \cdot (6 \cdot (n-1) + 1) = 6n^2 - 11n + 5$. Altogether the upper estimation of the number of moves to solve the first row and the first column is: $12n^2 - 13n + 5$.

Suppose that the position $(1,1)$ is unoccupied and let us denote $S(n)$ the estimation of the number of moves to solve the entire $(n^2 - 1)$-puzzle. Then it holds that: $S(n) = S(n-1) + 12n^2 - 13n + 5$, where $S(3) = 34$ (calculated as the upper for average length of optimal solutions). After solving the recurrent equation we obtain that: $S(n) = 4n^3 - \frac{1}{2}n^2 + \frac{3}{2}n - 70$. ■

Notice, that this is a new theoretical result for the Parberry's algorithm (in [6] only the worst case upper bound of $5n^3 + \mathcal{O}(n^2)$ and lower bounds are given).

Observation 1 ***(Average-case Solution Length – Snake-based).*** The average length of solutions to $(n^2 - 1)$-puzzle generated by the Snake-based algorithm is strictly lower than that of solutions generated by the algorithm of Parberry. ■

Commentary. The average length of the solution in random instances in the case of the snake-based algorithm can be expressed as the average number of moves necessary to place first two pebbles (top-down/left-right direction) plus the average solution length to instances where first two pebbles are already placed. Notice

that the average number of moves to place the first two pebbles is strictly lower in the snake-based algorithm. Hence, if we unfold the recurrence expression for the average length of the solution entirely the result will be strictly smaller than the average length of solutions of Parberry's algorithm. ■

Although we don't provide any explicit formula for the average length of solutions generated by our snake-base algorithm we know that it is strictly less than $4n^3 - \frac{1}{2}n^2 + \frac{3}{2}n - 70$.

Appendix B – BIBOX Algorithm with Snakes

A commented pseudo-code of the *BIBOX* algorithm enhanced with snakes is given in this appendix. The original *BIBOX* algorithm arranges robots into ears while the problem inductively reduces on a smaller biconnected graph whenever robots are arranged into the ear – robots in such an ear do not move any more. The algorithm is listed below as Algorithm 1. It uses several auxiliary functions to solve subtasks. Pseudo-code of auxiliary functions is given in [16] – here they are only briefly described.

The algorithm starts with constructing ear decomposition (line 1). It is assumed that a cycle denoted as $C(L_i)$ is associated with each ear; $C(L_i)$ can be constructed by adding a path connecting ear's connection vertices u and v. Then the goal configuration of robots is transformed so that vacant vertices are eventually located in the initial cycle of the decomposition (line 2). The algorithm solves this modified instance afterwards. The solution of the original instance is obtained by relocating vacant vertices from initial cycle to their original goal locations (line 8). This instance transformation is carried out by auxiliary functions *Transform-Goal* and *Finish-Solution* that relocates vacant vertices along two vertex disjoint paths. The main loop (lines 4-6) processes ear from the last one towards the initial cycle. Robots are arranged by another auxiliary procedure *Solve-Original-Cycle* in the original cycle (line 7).

Individual ears are processed by the procedure *Solve-Regular-Ear*. It arranges robots into the ear in stack like manner. First, unoccupied vertices are moved out of the processed ear as they will be needed there (lines 10-14). Then robots, whose goal positions are in the ear, are processed. Two cases are distinguished depending on whether the processed robot is located outside the ear (lines 17-25) or within the ear (lines 27-51). The easier case is with robot outside – in this case, the robot is moved to the connection vertex u using either *Move-Robot* or *Move-Robot-Snake* auxiliary procedure. The other connection vertex v is vacated by *Make-Unoccupied* procedure.

Algorithm 3. *The **BIBOX with snakes** algorithm.* The algorithm solves cooperative path-finding problem (CPF) over bi-connected graphs consisting of a cycle and at least one ear with two unoccupied vertices. The algorithm proceeds inductively according to the ear decomposition. The two unoccupied vertices are necessary for arranging robots within the initial cycle in the rest of the graph only one unoccupied vertex is needed. The pseudo-code is built around several higher-level operations. The modification from the original version consists in placing robots into an ear where two consecutive robots are considered at once. If consecutive robots are closed enough they are relocated towards the ear in a snake like manner together.

- Lock(U) — locks all the vertices from set U; each vertex is either *locked* or *unlocked*; an robot must not be moved out of the locked vertex which is respected by other operations
- Unlock(U) — unlocks all the vertices from set U
- Make-Unoccupied(v) — vacates vertex v
- Move-Robot(r, v) — moves robot r from its current location to vertex v
- Move-Robot-Snake(r, s, v) — moves robots r and s from their current locations towards v; that is r is moved to v and s is moved together with s in a snake-like manner if r and s are close enough initially ($\text{dist}_G(v, \alpha(r)) + \text{dist}_G(\alpha(r), \alpha(s)) < \text{dist}_G(v, \alpha(r)) + \text{dist}_G(v, \alpha(s))$; that is, the total distance towards destination v is smaller if robots go together than if they go one by one)
- Rotate-Cycle$^+(C)$ — rotates cycle C in the positive direction; a vacant vertex must be present in the cycle
- Rotate-Cycle$^-(C)$ — rotates cycle C in the negative direction
- Transform-Goal(G, R, α_+) — transforms goal configuration α_+ to a new configuration so that finally unoccupied vertices are located in the initial cycle of the ear decomposition; two disjoint paths along which empty vertices are relocated are returned
- Finish-Solution(φ, χ) — transforms the configuration with two unoccupied vertices in the initial cycle to the original goal configuration; φ and χ are two disjoint paths along which empty vertices shifted
- Solve-Original-Cycle — arranges robots within the initial cycle of the ear decomposition to comply with the transformed goal configuration; two empty vertices are employed to arrange robots

procedure *BIBOX-Snake-Solve*$(G = (V, E), R, \alpha_0, \alpha^+)$

/* Top level function of the BIBOX algorithm with snakes; solves a given cooperative path-finding problem.
Parameters: G - a graph modeling the environment,
R - a set of robots,
α_0 - a initial configuration of robots,
α_+ - a goal configuration of robots. */

```
1: let 𝒟 = [C_0, L_1, L_2, ..., L_k] be a ear decomposition of G
2: (α_+, φ, χ) ← Transform-Goal(G, R, α_+)
3: α ← α_0
4: for c = k, k − 1, ..., 1 do
5: |  if |L_c| > 2 then
6: |  |  Solve-Regular-Ear(c)
7: Solve-Original-Cycle
8: Finish-Solution(φ, χ)
```

---

procedure *Snake-Solve-Regular-Ear*(c)
/* Places robots which destinations are within a ear L_c; robots placed in the ear L_c are finally locked so they cannot move any more.
Parameters: c – index of a ear */
9: **let** $[u, w_1, w_2, \ldots, w_l, v] = L_c$
/* Both unoccupied vertices must be located outside the currently solved ear. */
10: **let** $x, z \in V \setminus \bigcup_{c=j}^{k} (L_j \setminus \{u, v\})$ such that $x \neq z$
11: *Make-Unoccupied*(x)
12: Lock$(\{x\})$
13: *Make-Unoccupied*(z)
14: Unlock$(\{x\})$
15: **for** $i = l, l-1, \ldots, 1$ **do**
16: *Lock*$(L_c \setminus \{u, v\})$
/* An robot to be placed is outside the ear L_c. */
17: **if** $\alpha(\alpha_+^{-1}(w_i)) \notin (L_c \setminus \{u, v\})$ **then**
18: **if** $i > 1$ **then**
19: *Move-Robot-Snake*$(\alpha_+^{-1}(w_i), \alpha_+^{-1}(w_{i-1}), u)$ — **modification** w.r.t. original *BIBOX*
20: **else**
21: *Move-Robot*$(\alpha_+^{-1}(w_i), u)$
22: Lock$(\{u\})$
23: *Make-Unoccupied*(v)
24: Unlock(L_c)
25: *Rotate-Cycle*$^+(C(L_c))$
/* An robot to be placed is inside the ear L_c. */
26: **else**
27: *Make-Unoccupied*(u)
28: Unlock(L_c)
29: $\rho \leftarrow 0$
30: **while** $\alpha(\alpha_+^{-1}(w_i)) \neq v$ **do**
31: *Rotate-Cycle*$^+(C(L_c))$
32: $\rho \leftarrow \rho + 1$
33: Lock$(L_c \setminus \{u, v\})$
34: **let** $y \in V \setminus (\bigcup_{j=c+1}^{d} (L_j \setminus \{u, v\}) \cup C(L_j))$
35: **if** $i > 1$ **then**
36: *Move-Robot-Snake*$(\alpha_+^{-1}(w_i), \alpha_+^{-1}(w_{i-1}), y)$ — **modification** w.r.t. original *BIBOX*
37: **else**
38: *Move-Robot* $(\alpha_+^{-1}(w_i), y)$
39: Lock $(\{y\})$
40: *Make-Unoccupied* (u)
41: Unlock (L_c)
42: **while** $\rho > 0$ **do**
43: *Rotate-Cycle*$^-(C(L_c))$
44: $\rho \leftarrow \rho - 1$
45: Unlock $(\{y\})$
46: Lock $(L_c \setminus \{u, v\})$
47: *Move-Robot* $(\alpha_+^{-1}(w_i), u)$

```
48: |   |   Lock ({u})
49: |   |   Make-Unoccupied (v)
50: |   |   Unlock (L_c)
51: |_  |_  Rotate-Cycle+(C(L_c))
52: Lock (L_c \ {u, v})
```

If some vertex is free on the cycle $C(L_c)$ then the cycle can be rotated which is done once in the positive direction by *Rotate-Cycle*$^+$ function. The rotation places the robot into the ear. Throughout the relocation of robots vertex locking is used (functions *Lock* and *Unlock*) to fix an robot in certain vertex while other robots or vacant vertex are relocated.

A more difficult case appears if the robot is inside the handle. In such case, the robot must be rotated out of the handle to the rest of the graph (lines 30-32). The number of positive rotations to get the robot out of the handle is counted (lines 27-32). The counted number of rotations is used to restore the situation by the corresponding number of negative rotations (lines 42 -44). At this point, the situation is the same as in the previous case. Thus, the robot is stacked into the handle in the same way.

The difference of *BIBOX* algorithm with snakes from the original *BIBOX* algorithm consists in adding *Move-Robot-Snake* procedure. When robots are relocated towards the currently processed ear, the snake based reasoning considers two consecutive robots whenever possible (lines 18-19 and 35-36). That is, while in the original algorithm, a single robot has been always relocated, in the snake version, the next to be relocated robot is considered as well. If both consecutive robots are close enough to each other they are relocated towards their target ear together in tandem (the process of tandem relocation is implemented within *Move-Robot-Snake* procedure).

Time Expansion Propositional Encodings for Makespan Optimal Solving of Cooperative Path Finding Problem

Pavel Surynek
Charles University in Prague
Faculty of Mathematics and Physics
Malostranské náměstí 25, 118 00 Praha 1, Czech Republic
pavel.surynek@mff.cuni.cz

Abstract. The problem of makespan optimal solving of cooperative path finding (CPF) is addressed in this paper. The task in CPF is to relocate a group of agents in a non-colliding way so that each agent eventually reaches its goal location from the given initial location. The abstraction adopted in this work assumes that agents are discrete items moving in an undirected graph by traversing edges. Makespan optimal solving of CPF means to generate solutions that are as short as possible in terms of the total number of time steps required for the execution of the solution.

We show that reducing CPF to propositional satisfiability (SAT) represents a viable option for obtaining makespan optimal solutions. Several encodings of CPF into propositional formulae are suggested and experimentally evaluated. The evaluation indicates that SAT based CPF solving outperforms other makespan optimal methods significantly in highly constrained situations (environments that are densely occupied by agents).

Keywords**:** cooperative path-finding (CPF), propositional satisfiability (SAT), time expanded graphs, makespan optimality, multi-robot path planning, multi-agent path finding, pebble motion on a graph

1. Introduction and Motivation

Cooperative path-finding - CPF [14, 23, 25] (also known as *multi-agent path finding* - MAPF [21, 22, 37, 38] or as *multi-robot path planning* - MRPP [18, 19] or as *pebble motion on a graph* - PMG [14, 16]) is an abstraction for many real-file tasks where the goal is to relocate some objects that spatially interacts with each other. In case of CPF, we are speaking about mobile *agents* (or robots) that can be

moved in a certain environment. Each agent starts at a given initial position in the environment and it is assigned a unique goal position to which it has to relocate. The problem consists in finding a spatial-temporal path for each agent by which the agent can relocate itself from its initial position to the given goal without colliding with other agents (that are simultaneously trying to reach their goals as well).

A graph theoretical abstraction, where the environment in which agents are moving is modeled as an undirected graph, is often adopted [18, 20]. Agents are represented as discrete items placed in vertices of the graph in this abstraction. Space occupancy imposed by presence of agents is modeled by the requirement that at most one agent resides in each vertex.

Movements of agents are also greatly simplified in the abstraction. An agent can instantaneously move to a neighboring vertex assumed that the target vertex is unoccupied and no other agent is trying to enter the same target vertex simultaneously. Note that various versions of the problem may have different conditions on movements - sometimes it is for instance allowed to move agents in a train like manner [28] or even rotate agents around cycle without any unoccupied vertex in the cycle [39].

There are many practical motivations for CPF ranging from unit navigation in computer games [24] to item relocation in automated storage (see KIVA robots [13]). Interesting motivations can be also found in traffic where problems like vessel avoidance at sea are of great practical importance [12]. An analogical challenge appears in the air where availability of drones implies need for developing cooperative air traffic control mechanisms [15].

We suggest to solve CPF via reducing it to *propositional satisfiability* (SAT) [7]. Particularly we are dealing with so-called *makespan optimal* solving of CPF [23, 29], which means to find a solution of a makespan as short as possible. The makespan of a solution is the number of steps necessary to execute all the moves of the solution. In other words, it is the length of the longest path from paths traveled by individual agents.

It is known that finding makespan optimal solutions to CPF is a difficult problem, namely it is NP-hard [16, 32, 39]. Hence reducing the makespan optimal CPF to SAT is justified as both problems are at the same level in terms of the complexity. Moreover, the reduction allows exploiting the power of modern SAT solvers [2, 3] in CPF solving. The question however is the design of an encoding of the CPF problem into propositional formula.

Several encodings of CPF into propositional formulae are introduced in this paper. They are based on a so-called *time expansion* of the graph that models the environment [11, 26] so that the formula can represent all the possible arrange-

ments of agents at all the time steps up to the given final time step. All the encodings are thoroughly experimentally evaluated with each other and also with alternative techniques for makespan optimal CPF solving.

2. Context of Related Works

The approach to solve CPF by reducing it to SAT has multiple alternatives. There exist algorithms based on search that find makespan optimal or near optimal solutions. The seminal work in this category is represented by Silver's WHCA* algorithm [20] which is a variant of A* search where cooperation among agents is incorporated. Recent contributions include OD+ID [23], which is a combination of A* and powerful agent independence detection heuristics, and ICTS [21] which employs the concept of increasing cost tree (instead of makespan, the total cost of solution is optimized). Other approaches resolve conflicts among robot trajectories when avoidance is necessary [5, 8, 34].

Fast polynomial time algorithms for generating makespan suboptimal solutions include PUSH-AND-ROTATE [37, 38] and other algorithms [28]. The drawback of these algorithms is that their solutions are dramatically far from the optimum.

Translation of CPF to a different formalism, namely to answer set programming (ASP), has been suggested in [9]. Integer programming (IP) as the target formalism has been also used [39]. The choice of SAT as the target formalism is very common in domain independent planning where the idea of time expansion [10, 11] and its reductions [4, 35] are studied.

3. Background

An arbitrary **undirected graph** can model the environment where agents are moving. Let $G = (V, E)$ be such a graph where $V = \{v_1, v_2, \dots, v_n\}$ is a finite set of vertices and $E \subseteq \binom{V}{2}$ is a set of edges. The configuration of agents in the environment is modeled by assigning them vertices of the graph. Let $A = \{a_1, a_2, \dots, a_\mu\}$ be a finite set of *agents*. Then, a configuration of agents in vertices of graph G will be fully described by a *location* function $\alpha: A \longrightarrow V$; the interpretation is that an agent $a \in A$ is located in a vertex $\alpha(a)$. At most **one agent** can be located in a vertex; that is α is a uniquely invertible function. A generalized inverse of α denoted as $\alpha^{-1}: V \longrightarrow A \cup \{\bot\}$ will provide us an agent located in a given vertex or $\bot$ if the vertex is empty.

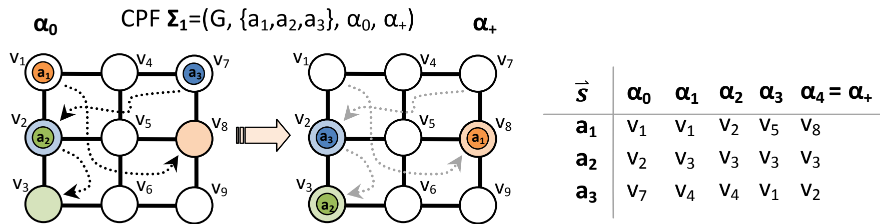
Definition 1 (COOPERATIVE PATH FINDING). An instance of *cooperative path-finding* problem (CPF) is a quadruple $\Sigma = [G = (V, E), A, \alpha_0, \alpha_+]$ where location functions α_0 and α_+ define the initial and the goal configurations of a set of agents A in G respectively. □

The dynamicity of the model assumes a discrete time divided into time steps. A configuration α_i at the i-th time step can be transformed by a transition action which instantaneously moves agents in the non-colliding way to form a new configuration α_{i+1}. The resulting configuration α_{i+1} must satisfy the following *validity conditions*:

- $\forall a \in A$ either $\alpha_i(a) = \alpha_{i+1}(a)$ or $\{\alpha_i(a), \alpha_{i+1}(a)\} \in E$ holds (agents move along edges or not move at all), (1)
- $\forall a \in A \;\; \alpha_i(a) \neq \alpha_{i+1}(a) \Rightarrow \alpha_i^{-1}(\alpha_{i+1}(a)) = \bot$ (agents move to vacant vertices only), and (2)
- $\forall a, b \in A \;\; a \neq b \Rightarrow \alpha_{i+1}(a) \neq \alpha_{i+1}(b)$ (no two agents enter the same target/unique invertibility of resulting arrangement). (3)

The task in cooperative path finding is to transform α_0 using above valid transitions to α^+. An illustration of CPF and its solution is depicted in Figure 1.

Definition 2 (SOLUTION, MAKESPAN). A *solution* of a *makespan* η to a cooperative path finding instance $\Sigma = [G, A, \alpha_0, \alpha^+]$ is a sequence of arrangements $\vec{s} = [\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_\eta]$ where $\alpha_\eta = \alpha^+$ and α_{i+1} is a result of valid transformation of α_i for every $i = 1,2, \ldots, \eta - 1$. □



$\vec{s}$	α_0	α_1	α_2	α_3	$\alpha_4 = \alpha_+$
a_1	v_1	v_1	v_2	v_5	v_8
a_2	v_2	v_3	v_3	v_3	v_3
a_3	v_7	v_4	v_4	v_1	v_2

Figure 1. *An example of cooperative path-finding problem (CPF).* Three agents a_1, a_2, and a_3 need to relocate from their initial positions represented by α_0 to goal positions represented by α_+. A solution of makespan 4 is shown.

The number $|\vec{s}| = \eta$ is a *makespan* of solution $\vec{s}$. It is often a question whether there exists a solution of Σ of the given makespan $\eta \in \mathbb{N}$. This is known as a *deci-*

sion variant of CPF. It is known that the decision variant of CPF is NP-complete, hence finding makespan optimal solution to CPF is NP-hard [16]. Note that due to no-ops introduced in valid transitions, it is equivalent to ask whether there is a solution of exactly the given makespan ant to ask whether there is a solution of at most given makespan.

4. Solving CPF Optimally through Propositional Satisfiability

The question we are addressing is how to obtain *makespan optimal solutions* of CPFs in some practical manner. The approach we are suggesting here employs *propositional satisfiability* (SAT) [1] solving as the key technology. Note that the decision variant of CPF is in NP, hence it can be reduced to propositional satisfiability. That is, a propositional formula $F(\Sigma,\eta)$ such that it is satisfiable if and only if a given CPF Σ with makespan η is solvable can be constructed. Being able to construct such a formula $F(\Sigma,\eta)$ one can obtain the optimal makespan for the given CPF Σ by asking multiple queries whether formula $F(\Sigma,\eta)$ is satisfiable with different makespan bounds η.

Various strategies of choice of makespan bounds for queries exist for getting the optimal makespan. The simplest and efficient one at the same time is to try sequentially makespan $\eta = 1,2,\ldots$ until η equal to the optimal makespan is reached. This strategy will be further referred as *sequential increasing*. The sequential increasing strategy is also used in domain independent planners such as SATPLAN [11], SASE [10] and others. Pseudo-code of the strategy is listed as Algorithm 1.

Algorithm 1. *SAT-based optimal CPF solving – sequential increasing strategy.* The algorithm sequentially finds the smallest possible makespan η for that a given CPF $\Sigma = (G, A, \alpha_0, \alpha_+)$ is solvable. A question whether a solution of CPF Σ exists is constructed with respect to increasing makespans and submitted to a SAT solver.

input: Σ – a CPF instance
output: a pair consisting of the optimal makespan and corresponding optimal solution

---

function *Find-Optimal-Solution-Sequentially* $(\Sigma = (\mathrm{G}, A, \alpha_0, \alpha_+))$: **pair**
1: $\eta \leftarrow 1$
2 **loop**
3: $F(\Sigma,\eta) \leftarrow$*Encode-CPF-as-SAT* (Σ,η)
4: **if** *Solve-SAT* $(F(\Sigma,\eta))$ **then**
5: $s \leftarrow$ *Extract-Solution-from-Valuation*$(F(\Sigma,\eta))$
6: **return** (η, s)
7: $\eta \leftarrow \eta + 1$
8: **return** $(\infty, \emptyset)$

The focus here is on SAT encoding while querying strategies are out of scope of the paper; though let us mention that in depth study of querying strategies is given in [17]. There is a great potential in querying strategies as they can bring speedup of planning process in orders of magnitude, especially when combined with parallel processing.

The important property of propositional encoding $F(\Sigma, \eta)$ is that a solution of CPF Σ of makespan η can be unambiguously extracted from satisfying valuation of $F(\Sigma, \eta)$ (otherwise, equivalence between solvability of CPF Σ bounded by η and solvability of $F(\Sigma, \eta)$ could be trivially established by setting $F(\Sigma, \eta) \equiv TRUE$ in case Σ is solvable in η time steps and $F(\Sigma, \eta) \equiv FALSE$ otherwise).

Note that the solving process represented by Algorithm 1 is incomplete, as it does not terminate when the input instance is unsolvable. Nevertheless, the solving process can be easily made complete by checking instance solvability prior to SAT-based optimization by some fast polynomial time algorithm such as those described in [14, 28, 38].

The important advantage of solving CPF as SAT is that there exist many powerful solvers for SAT [2, 3] implementing numerous advanced techniques such as intelligent search space pruning and learning. The spectrum of these techniques is so rich and so well engineered in modern SAT solvers that it is almost impossible to reach the equal level of advancement in solving CPF by own dedicated solver. Nevertheless, all the well-engineered techniques implemented in SAT solvers can be employed in CPF solving if it is translated to SAT. Note, that the effect of SAT solving techniques is indirect in CPF solving as it is mediated through the translation. Hence, the design of the encoding of CPF as SAT should take into consideration the way in which SAT solvers operate.

4.1. Time Expansion Graphs

The *trajectory* of an agent in time over G is not necessarily *simple* in general case (that is, a single vertex can be visited multiple times). In a propositional representation of such kind of trajectory, it is difficult to fix the number of variables. Therefore, a graph derived from G by expanding it over time, where the trajectory of each agent will correspond to a simple path in this graph, will be used (a simple path visits each vertex of the graph at most once). The graph of required properties is introduced in the following definition and illustrated in Figure 3.

Definition 3 (TIME EXPANSION GRAPH - $\text{Exp}_{\text{T}}(G, \eta)$). Let $G = (V, E)$ be an undirected graph and $\eta \in \mathbb{N}$. A *time expansion graph* with $\eta + 1$ time layers (indexed from 0 to η) associated with G is a directed graph $\text{Exp}_{\text{T}}(G, \eta) = (V \times$

$\{0,1,\ldots,\eta\}, E')$ where $E^{'} = \{([u,l],[v,l+1]) \mid \{u,v\} \in E;\ l = 0,1,\ldots,\eta-1\} \cup \{([v,l],[v,l+1]) \mid v \in V; l = 0,1,\ldots,\eta-1\}$. □

Notation u^l will be sometimes used instead of $[u,l]$ in figures. The search for a solution of CPF with makespan bound η can be viewed as the search for a collection of so-called *non-overlapping vertex disjoint paths* in the corresponding time expansion graph consisting of η layers $\mathrm{Exp}_{\mathrm{T}}(G,\eta)$. This is also the reason why the number of time layers in time expansion graphs and the makespan bound in CPF use the same notation with η. Non-overlapping vertex disjoint paths must have disjoint set of endpoints of non-trivial edges in consecutive time layers of $\mathrm{Exp}_{\mathrm{T}}(G,\eta)$ as described in the following definition.

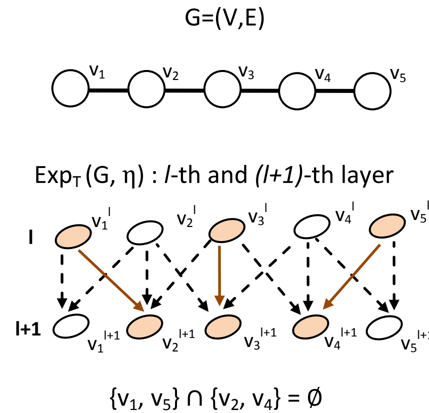


Figure 2. *An illustration of non-overlapping vertex disjoint paths*. Parts of three non-overlapping paths between time layers l and $l+1$ of $\mathrm{Exp}_{\mathrm{T}}(G,\eta)$ are shown.

Definition 4 (NON-OVERLAPPING VERTEX DISJOINT PATHS IN $\mathrm{Exp}_{\mathrm{T}}(G,\eta)$). A collection of paths $\Pi = \{\pi_1, \pi_2, \ldots, \pi_\mu\}$ in $\mathrm{Exp}_{\mathrm{T}}(G,\eta)$ so that π_i connects $[x_i, 0]$ with $[y_i, \eta]$ with $x_i, y_i \in V$ for $i = 1,2,\ldots,\mu$ is called to be *non-overlapping vertex disjoint* if and only if $\pi_i \cap \pi_j = \emptyset$ for any two $i,j \in \{1,2,\ldots,\mu\}$ with $i \neq j$ and $\{\pi_i[l,2] \mid \pi_i[l,2] \neq \pi_i[l+1,2] \wedge i = 1,2,\ldots,\mu\} \cap \{\pi_i[l+1,2] \mid \pi_i[l,2] \neq \pi_i[l+1,2] \wedge i = 1,2,\ldots,\mu\}$[1] for $l = 0,1,\ldots,\eta-1$. □

Non-overlapping vertex disjoint paths between two consecutive time layers of $\mathrm{Exp}_{\mathrm{T}}(G,\eta)$ are shown in Figure 2. The correspondence between existence of a solution to CPF and non-overlapping vertex disjoint paths is established in the next proposition.

Proposition 1 ***(NON-OVERLAPPING VERTEX DISJOINT PATHS IN EXP$_T$).*** *A solution of makespan* $\eta \in \mathbb{N}$ *of a CPF* $\Sigma = (G, A, \alpha_0, \alpha_+)$ *with* $A =$

[1] The notation $\pi_i[l,2]$ refers to the second component of the l-th element of π_i.

$\{a_1, a_2, \ldots, a_\mu\}$ exists if and only if there exist a set $\Pi = \{\pi_1, \pi_2, \ldots, \pi_\mu\}$ of non-overlapping vertex disjoint paths in $\text{Exp}_\text{T}(G, \eta)$ so that π_i connects $[\alpha_0(a_i),0]$ with $[\alpha_+(a_i), \eta]$ for $i = 1,2, \ldots, \mu$. ■

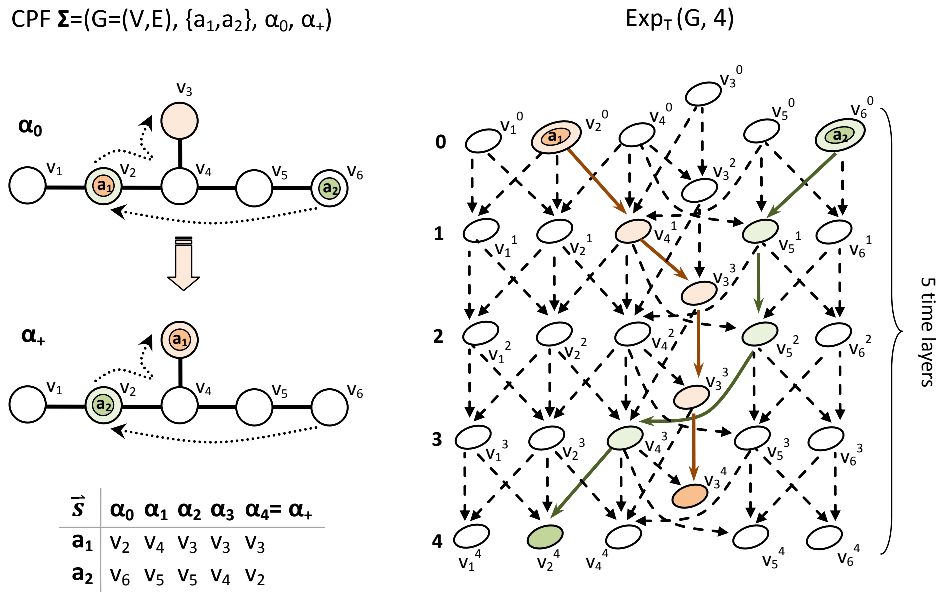


Figure 3. *An example of CPF and its time expansion graph.* A time expansion graph $\text{Exp}_\text{T}(G, 4)$ consisting of 5 time layers is build for a given CPF Σ. Solving Σ in 5 time steps can be represented as searching for a collection of non-overlapping vertex disjoint paths connecting the initial positions agents in the first layer with their goal positions in the last layer of $\text{Exp}_\text{T}(G, 4)$.

Proof. Assume that a solution $\vec{s} = [\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_\eta]$ of makespan η of given CPF Σ exists. Then vertex disjoint paths $\pi_1, \pi_2, \ldots, \pi_\mu$ in $\text{Exp}_\text{T}(G, \eta)$ can be constructed from $\vec{s}$. Path π_i will correspond to the trajectory of agent a_i; that is, $\pi_i = ([\alpha_0(a_i),0], [\alpha_1(a_i),1], \ldots, [\alpha_\eta(a_i), \eta])$. The path constructed in this way is a correct path in $\text{Exp}_\text{T}(G, \eta)$, since $\{\alpha_l(a_i), \alpha_{l+1}(a_i)\} \in E$ or $\alpha_l(a_i) = \alpha_{l+1}(a_i)$ for $l = 0,1, \ldots, \eta - 1$; that is, $([\alpha_l(a_i), l], [\alpha_{l+1}(a_i), l + 1]) \in E'$ holds by construction of $\text{Exp}_\text{T}(G, \eta)$. Obviously π_i connects $[\alpha_0(a_i),0]$ with $[\alpha_+(a_i) = \alpha_\eta(a_i), \eta]$ in $\text{Exp}_\text{T}(G, \eta)$. It remains to check that no two constructed paths intersect and that paths are non-overlapping. Validity condition (3) ensures that no two path share a common vertex since otherwise agents would collide. Validity conditions (1) and (2) together ensure that overlapping between set of endpoints of edges of paths between consecutive time layers happens only with trivial edges – that is, edges that continues into the same vertex in the next time layer.

Let us show the opposite implication. Assume that non-overlapping vertex disjoint paths $\pi_1, \pi_2, \ldots, \pi_\mu$ in $\mathrm{Exp_T}(G, \eta)$ exist. We will construct a solution of CPF Σ of makespan η. Assume that Let $\pi_i = ([u_0, 0], [u_1, 1], [u_2, 2], \ldots, [u_\eta, \eta])$, $u_l \in V$ for $l = 0,1, \ldots, \eta$ where $u_0 = \alpha_0(a_i)$ and $u_\eta = \alpha_+(a_i)$. The trajectory of agent a_i is set as follows: $\alpha_0(a_i) = u_0$, $\alpha_1(a_i) = u_1$, $\alpha_2(a_i) = u_2$, ..., $\alpha_\eta(a_i) = u_\eta$. It can be easily verified that validity conditions (1) – (3) are satisfied by such a construction. Paths are vertex disjoint, so agents do not collide by following them – condition (2) is satisfied. As paths do not overlap agents either stay in a vertex or move into a vertex that was not occupied in the previous step. Altogether, validity conditions (1) – (3) are satisfied. ■

4.2. Propositional Encodings Based on Time Expansion Graphs

The concept of time expansion graph represents an important step towards the design of a propositional formula that is satisfiable if and only if the given CPF has a solution of a given makespan. Moreover, we require such a formula where a corresponding CPF solution can be extracted from its satisfying valuation. Time expansion graph can be used as a basis for such a formula as it can capture all the arrangements of agents over the graph modeling the environment at all the time steps up to the given final step.

4.2.1. INVERSE Propositional Encoding

Let $\deg_G(v)$ denote the degree of vertex v in G; that is, $\deg_G(v)$ is the number of edges from E incident with v. It is further assumed that neighbors of each vertex v in G are assigned ordering numbers by a one-to-one assignment $\sigma_v: \{u | \{v, u\} \in E\} \longrightarrow \{1,2, \ldots, \deg_G(v)\}$ (that is, for each neighbor u of v we are told that it is a $\sigma_v(u)$-th neighbor). An inverse σ_v^{-1} is naturally defined (that is, $\sigma_v^{-1}(i)$ returns i-th neighbor of v for $i \in \{1,2, \ldots, \deg_G(v)\}$).

The following definition introduces the INVERSE encoding over finite domain state variables that will be further encoded into bit-vectors using the standard *binary encoding*.

Definition 5 (INVESE ENCODING – $F_{INV}(\eta, \Sigma)$). Assume that a CPF $\Sigma = [G, A, \alpha_0, \alpha_+]$ with $G = (V, E)$ is given. An *INVERSE encoding* for CPF Σ consists of the following finite domain variables for each time layer $l \in \{0,1, \ldots, \eta\}$: $\mathcal{A}_v^l \in \{0,1, \ldots, \mu\}$ for every $v \in V$ to model agent occurrences in vertices. For time layers $l \in \{0,1, \ldots, \eta - 1\}$ there are also finite domain variables $\mathcal{T}_v^l \in \{0,1, \ldots, 2 \cdot \deg_G(v)\}$ for every $v \in V$ to represent agent movements. Constraints of INVERSE encoding are as follows:

- $\mathcal{T}_v^l = 0 \Rightarrow \mathcal{A}_v^{l+1} = \mathcal{A}_v^l$ for every $v \in V$ and $l \in \{0,1,\ldots,\eta-1\}$ (4)
  (if there is no movement occurs in a vertex then the vertex hold the same agent at the next time step)
- $0 < \mathcal{T}_v^l \leq \deg_G(v) \Rightarrow \mathcal{A}_u^l = 0 \wedge \mathcal{A}_u^{l+1} = \mathcal{A}_v^l \wedge \mathcal{T}_u^l = \sigma_u(v) + \deg_G(u)$, (5)
  for every $v \in V$ and $l \in \{0,1,\ldots,\eta-1\}$, where $u = o_v^{-1}(\mathcal{T}_v^l)$
  (an agent leaves from v to its $\mathcal{T}_v^l$-th neighbor u)
- $\deg_G(v) < \mathcal{T}_v^l \leq 2 \cdot \deg_G(v) \Rightarrow \mathcal{T}_u^l = \sigma_u(v)$,
  for every $v \in V$ and $l \in \{0,1,\ldots,\eta-1\}$, where $u = \sigma_v^{-1}(\mathcal{T}_v^l - \deg_G(v))$ (6)
  (an agent leaves arrives to v from its $(\mathcal{T}_v^l - \deg_G(v))$-th neighbor u). □

Initial and goal arrangements will be expressed though the following constraints:

- $\mathcal{A}_u^0 = i$ for $u \in V$ if there is $i \in \{1,2,\ldots,\mu\}$ such that $\alpha_0(a_i) = u$ (7) — Initial locations
- $\mathcal{A}_u^0 = 0$ for $u \in V$ if $(\forall a \in A)\alpha_0(a) \neq u$ (8) — Initial locations
- $\mathcal{A}_u^\eta = i$ for $u \in V$ if there is $i \in \{1,2,\ldots,\mu\}$ such that $\alpha_+(a_i) = u$ (9) — Goal locations
- $\mathcal{A}_u^\eta = 0$ for $u \in V$ if $(\forall a \in A)\alpha_+(a) \neq u$ (10) — Goal locations

The resulting propositional formula in CNF, where $\mathcal{A}_v^l$ and $\mathcal{T}_v^l$ variables are replaced with bit vectors with binary encoding and constraints are replaced accordingly, will be denoted as $F_{INV}(\eta, \Sigma)$.

The meaning of $\mathcal{A}_v^l$ variables correspond to the inverse location function at time step l. That is, if the inverse location function at time step l is α_l^{-1} then $\mathcal{A}_v^l = j$ iff $\alpha_l^{-1}(v) = a_j$ and $\mathcal{A}_v^l = 0$ iff $\alpha_l^{-1}(v) = \perp$. Variables $\mathcal{T}_v^l$ represent transitions of agents among vertices. Zero value is reserved for no-movement. Half of remaining values from 1 to $\deg_G(v)$ represent outgoing movements from v to some neighbor indicated by $\mathcal{T}_v^l$; the other half of values represent incoming movements into v from some of its neighbors indicated by $\mathcal{T}_v^l - \deg_G(v)$.

It is not straightforward to encode the above finite domain model into propositional model where finite domain state variables are replaced with *bit-vectors* (vectors of propositional variables) using *binary encoding* as we need to represent quite complex integer constraints over bit vectors. Variables $\mathcal{A}_v^l$ are modeled by a vector of $\lceil \log_2(\mu+1) \rceil$ propositional variables where individual (propositional) bits will be accessed by a bit index $\mathbb{i} \in \{0,1,\ldots,\lceil \log_2(\mu+1) \rceil - 1\}$ denoted as $\mathcal{A}_v^l[\mathbb{i}]$. Variables $\mathcal{T}_v^l$ are modeled by vectors of $\lceil \log_2(2 \cdot \deg_G(v) + 1) \rceil$ propositional variables. Note, that typical environments are connected only locally, which

means that $\deg_G(v) \ll \mu$ typically. If the represented finite domain variable has the number of states that is different from the power of 2, then extra states are forbidden.

Constraints need to distinguish between all the $2 \cdot \deg_G(v) + 1$ states of $\mathcal{T}_v^l$ variables since over bit vectors we are able to express very simple constraints only – such as an expression that a bit vector equals to a constant. Note that over $\mathcal{A}_v^l$ variables we only need to model equality between them and equality to zero which does not distinguish between too many cases. Let $\mathbb{b}: \mathbb{N}_0 \times \mathbb{N}_0 \to \{0,1\}$ be a binary representation of positive integers where $\mathbb{b}(x, \mathbb{i})$ represents value of the $\mathbb{i}$-th bit in binary encoding of x; that is $x = \sum_{\mathbb{i}=0}^{b-1} \mathbb{b}(x, \mathbb{i}) \cdot 2^{\mathbb{i}}$.

Equality of a $\mathcal{T}_v^l$ variable to a given constant $c \in \{0,1, ..., 2 \cdot \deg_G(v)\}$ will be expressed as following conjunction:

$$\text{con}_=(\mathcal{T}_v^l, c) = \bigwedge_{\mathbb{i}=0}^{\lceil \log_2(2 \cdot \deg_G(v)+1) \rceil - 1} \text{lit}(\mathcal{T}_v^l, c, \mathbb{i})$$

$$\text{where } \text{lit}(\mathcal{T}_v^l, c, \mathbb{i}) = \begin{cases} \mathcal{T}_v^l[\mathbb{i}] & \text{iff } \mathbb{b}(c, \mathbb{i}) = 1 \\ \neg\mathcal{T}_v^l[\mathbb{i}] & \text{iff } \mathbb{b}(c, \mathbb{i}) = 0 \end{cases} \quad (11)$$

Equality between variables $\mathcal{A}_v^l$ and $\mathcal{A}_u^{l+1}$ is expressed by the following conjunction of equivalences:

$$\text{var}_=(\mathcal{A}_v^l, \mathcal{A}_u^{l+1}) = \bigwedge_{\mathbb{i}=0}^{\lceil \log_2(\mu+1) \rceil - 1} (\neg\mathcal{A}_v^l[\mathbb{i}] \vee \mathcal{A}_u^{l+1}[\mathbb{i}]) \wedge (\mathcal{A}_v^l[\mathbb{i}] \vee \neg\mathcal{A}_u^{l+1}[\mathbb{i}]) \quad (12)$$

The above elementary constructions are put together to represent constraints (4) – (6) using Tseitin's encoding [33] which introduces auxiliary propositional variables to the encoding. Auxiliary propositional variables $a_{v,l}^{\text{zero}}$ representing empty vertex v at time step l, $a_{u,v,l}^{=}$ representing equality between $\mathcal{A}_v^l$ and $\mathcal{A}_u^{l+1}$, and $a_{v,l,c}^{\text{tran}}$ representing equality $\mathcal{T}_v^l = c$. The connection of auxiliary variables with their exact meaning is done by the following constraints:

$$a_{v,l}^{\text{zero}} \Rightarrow \text{con}_=(\mathcal{A}_v^l, 0) \quad (13)$$

$$a_{u,v,l}^{=} \Rightarrow \text{var}_=(\mathcal{A}_v^l, \mathcal{A}_u^{l+1}) \quad (14)$$

$$a_{v,l,c}^{\text{tran}} \Leftrightarrow \text{con}_=(\mathcal{T}_v^l, c) \quad (15)$$

As $\mathcal{A}_v^l$ variables appear only on the right side of implications in constraints (4) – (6) of the INVERSE encoding it is sufficient to connect their auxiliary by impli-

cations only. Whereas $\mathcal{T}_v^l$ variables appear on both sides of implications in (4) – (6); therefore they need to be connected by equivalences to their auxiliary variables.

Having above auxiliary variables, INVERSE encoding constraints can be easily expressed using them as follows:

- $a_{v,l,0}^{\text{tran}} \Rightarrow a_{v,v,l}^{=}$ (16)
  for every $v \in V$ and $l \in \{0,1,\ldots,\eta-1\}$
- $a_{v,l,c}^{\text{tran}} \Rightarrow a_{u,l}^{\text{zero}} \wedge a_{u,v,l}^{=} \wedge a_{u,l,\sigma_u(v)+\deg_G(u)}^{\text{tran}}$ (17)
  for each $0 < c \leq \deg_G(v)$, $v \in V$ and $l \in \{0,1,\ldots,\eta-1\}$,
  where $u = o_v^{-1}(c)$
- $a_{v,l,c}^{\text{tran}} \Rightarrow a_{u,l,\sigma_u(v)}^{\text{tran}}$ (18)

  for each $\deg_G(v) < c \leq 2 \cdot \deg_G(v)$, $v \in V$ and $l \in \{0,1,\ldots,\eta-1\}$, where $u = \sigma_v^{-1}\big(\mathcal{T}_v^l - \deg_G(v)\big)$

In the following space consumption of the INVERSE encoding only regular time layers are counted as asymptotically requirements of the initial and final time layers are dominated by the rest.

Proposition 2 ***(INVERSE ENCODING SIZE).*** *The number of visible propositional variables in* $F_{INV}(\eta,\Sigma)$ *is* $\mathcal{O}\big(\eta \cdot (|V| \cdot \lceil\log_2(\mu)\rceil + \sum_{v\in V}\lceil\log_2(\deg_G(v))\rceil)\big)$ *and there are* $\mathcal{O}(\eta \cdot (|V| + |E|))$ *auxiliary variables; that is* $\mathcal{O}\big(\eta \cdot (|V| \cdot \lceil\log_2(\mu)\rceil + \sum_{v\in V}\lceil\log_2(\deg_G(v))\rceil + |E|)\big)$ *propositional variables in total. The number of clauses is* $\mathcal{O}\big(\eta \cdot (|V| \cdot \lceil\log_2(\mu)\rceil + |E| \cdot \lceil\log_2(\mu)\rceil + \sum_{v\in V}\deg_G(v) \cdot (\lceil\log_2(\deg_G(v))\rceil))\big)$. ■

Proof. To show the result we need just to calculate variables and clauses. The visible variables, that is, propositional variables representing $\mathcal{A}_v^l$ and $\mathcal{T}_v^l$ counts for $(\eta+1) \cdot |V| \cdot \lceil\log_2(\mu+1)\rceil$ and $\eta \cdot \sum_{v\in V}\lceil\log_2(2 \cdot \deg_G(v)+1)\rceil$ respectively. The number of auxiliary variables $a_{v,l}^{\text{zero}}$ is $(\eta+1) \cdot |V|$; the number of $a_{u,v,l}^{=}$ variables is $(\eta+1) \cdot |E|$; and the number of $a_{v,l,c}^{\text{tran}}$ variables is $2 \cdot \eta \cdot \sum_{v\in V}\deg_G(v)$ which is $4 \cdot \eta \cdot |E|$. Hence the total number of propositional variables is $(\eta+1) \cdot (|V| \cdot \lceil\log_2(\mu+1)\rceil + |V| + |E|) + \eta \cdot (\sum_{v\in V}\lceil\log_2(2 \cdot \deg_G(v)+1)\rceil + 4 \cdot |E|)$ which is $\mathcal{O}\big(\eta \cdot (|V| \cdot \lceil\log_2(\mu)\rceil + \sum_{v\in V}\lceil\log_2(\deg_G(v))\rceil + |E|)\big)$.

Let us calculate the number of clauses. A single constraint (13) develops into $\lceil\log_2(\mu+1)\rceil$ binary clauses; a single constraint (14) develops into $2\cdot\lceil\log_2(\mu+1)\rceil$ ternary clauses; and a single constraint (15) develops into $\lceil\log_2(2\cdot\deg_G(v)+1)\rceil$ binary clauses and one clause of arity $\lceil\log_2(2\cdot\deg_G(v)+1)\rceil+1$. There is as many as $\eta\cdot|V|$ constraints (13); $\eta\cdot|E|$ constraints (14); and $\eta\cdot\sum_{v\in V}\deg_G(v)$ constraints (15) which in total gives $\eta\cdot((|V|+2\cdot|E|)\cdot\lceil\log_2(\mu+1)\rceil+\sum_{v\in V}\deg_G(v)\cdot(\lceil\log_2(2\cdot\deg_G(v)+1)\rceil+1))$ clauses (binary, ternary, and one multi-arity).

Constraints (16) count for $\eta\cdot|V|$ binary clauses, constraints (17) together with (18) count for $4\cdot\eta\cdot\sum_{v\in V}\deg_G(v)$ binary clauses which is clearly dominated by the already calculated number of clauses. Hence, we have $\eta\cdot(|V|\cdot\lceil\log_2(\mu)\rceil+|E|\cdot\lceil\log_2(\mu)\rceil+\sum_{v\in V}\deg_G(v)\cdot(\lceil\log_2(\deg_G(v))\rceil))$ clauses. ■

Proposition 3 ***(PATHS AND*** $F_{INV}(\eta,\Sigma)$ ***SATISFACTION).*** *A set* $\Pi=\{\pi_1,\pi_2,\dots,\pi_\mu\}$ *of non-overlapping vertex disjoint paths in* $\text{Exp}_\text{T}(G,\eta)$ *so that* π_i *connects* $[\alpha_0(a_i),0]$ *with* $[\alpha_+(a_i),\eta]$ *for* $i=1,2,\dots,\mu$ *exists if and only if* $F_{INV}(\eta,\Sigma)$ *is satisfiable. Moreover, paths* $\pi_1,\pi_2,\dots,\pi_\mu$ *can be unambiguously constructed from satisfying valuation of* $F_{INV}(\eta,\Sigma)$ *and vice versa.* ■

Sketch of proof. For simplicity, we will show the proposition over finite domain variables instead of bit-vectors. The equivalence between bit vectors and finite domain variables is can be seen directly from the translation of finite domain constraints to equivalent constraints over bit vectors.

Assume that there exists a collection of vertex disjoint paths $\Pi=\{\pi_1,\pi_2,\dots,\pi_\mu\}$, where π_i connects $[\alpha_0(a_i),0]$ with $[\alpha_+(a_i),\eta]$. Let $\pi_i=([u_0,0]$, $[u_1,1]$, $[u_2,2]$, ..., $[u_\eta,\eta])$, $u_l\in V$ for $l=0,1,\dots,\eta$ where $u_0=\alpha_0(a_i)$ and $u_\eta=\alpha_+(a_i)$. We can set $\mathcal{A}_{u_0}^0=i$, $\mathcal{A}_{u_1}^1=i$, ..., $\mathcal{A}_{u_\eta}^\eta=i$. Transition variables are set according to traversed edges; that is, $\mathcal{T}_{u_0}^0=\sigma_{u_0}(u_1)$, $\mathcal{T}_{u_1}^0=\sigma_{u_1}(u_0)+\deg_G(u_1)$, $\mathcal{T}_{u_1}^1=\sigma_{u_1}(u_2)$, $\mathcal{T}_{u_2}^1=\sigma_{u_2}(u_1)+\deg_G(u_2)$, ..., $\mathcal{T}_{u_l}^l=\sigma_{u_l}(u_{l+1})$, $\mathcal{T}_{u_{l+1}}^l=\sigma_{u_{l+1}}(u_l)+\deg_G(u_{l+1})$, ..., $\mathcal{T}_{u_{\eta-1}}^{\eta-1}=\sigma_{u_{\eta-1}}(u_\eta)$, $\mathcal{T}_{u_\eta}^{\eta-1}=\sigma_{u_\eta}(u_{\eta-1})+\deg_G(u_\eta)$. Other paths from Π are processed in the same way. Observe that there is no conflict in setting the variables; that is, each variable is set at most once by the assignment; which is due to the fact that paths are vertex disjoin. Variables $\mathcal{A}_v^l$ and $\mathcal{T}_v^l$ that has not been set so far are set to 0. It is not difficult to check that constraints (4) – (6) as well as (7) – (11) are satisfied.

On the other hand, if there is a satisfying valuation of $F_{\eta-INV}(\Sigma)$ then we are able to reconstruct required vertex disjoint paths from it. Let $\pi_i=([u_0,0]$, $[u_1,1]$, $[u_2,2]$, ..., $[u_\eta,\eta])$ where $u_0=\alpha_0(a_i)$, and $u_{l+1}=\sigma_{u_l}^{-1}(\mathcal{T}_{u_l}^l)$ for every

$l = 0,1, \dots, \eta - 1$ (it holds also that $u_l = \sigma^{-1}_{u_{l+1}}(\mathcal{T}^{l+1}_{u_{l+1}}) - \deg_G(u_{l+1})$). Transition state variables $\mathcal{T}^l_v$ that take just one value ensure that each vertex at each time layer needs to decide if it either is connected to a neighbor or accepts a connection from a neighbor (or is connected to itself). It is ensured that no intersection between selected paths appears as otherwise a vertex must have accepted connections from at least two sources or has to branch connections to at least two neighbors, which is both forbidden. A value of $\mathcal{A}^l_v$ variable is propagated to the next time layer only through the connection of the corresponding transition state variable $\mathcal{T}^l_v$. The fact that agents were propagated to their goals ensures that there must be a paths induced by transition state variables from initial positions of agents to their goal. ■

The following theorem can be directly obtained by applying Proposition 1 and Proposition 3 which together justify solving of CPF via translation to SAT.

Theorem 1 ***(SOLUTION OF*** **Σ** ***AND*** **$F_{INV}(\eta, \Sigma)$** ***SATISFACTION).*** *A solution of a CPF $\Sigma = (G, A, \alpha_0, \alpha_+)$ with $A = \{a_1, a_2, \dots, a_\mu\}$ exists if and only if there ist $\eta \in \mathbb{N}$ for that formula $F_{\eta-INV}(\Sigma)$ is satisfiable.* ■

4.2.2. ALL-DIFFERENT Propositional Encoding

Choosing location function instead of its inverse for representing arrangements of agents at individual time steps led to another encoding called ALL-DIFFERENT – the name comes from the fact that it is necessary to express the requirement that each vertex is occupied by at most one agent explicitly which is modeled by pairwise differences between variables representing the arrangement. Again it is easier to express the encoding over finite domain state variables before it is transformed to propositional formula.

Definition 6 (ALL-DIFFERENT ENCODING – $F_{DIFF}(\eta, \Sigma)$). Assume that a CPF $\Sigma = [G, A, \alpha_0, \alpha_+]$ with $G = (V, E)$ is given. An *ALL-DIFFERENT encoding* for CPF Σ consists of finite domain variables $\mathcal{L}^l_a \in \{1, \dots, n\}$ for every $a \in A$ and each time layer $l \in \{0,1, \dots, \eta\}$ to model locations of agents over time. Constraints are as follows: (19)

- $\mathcal{L}^l_a = j \Rightarrow \mathcal{L}^{l+1}_a = j \vee \bigvee_{\jmath \in \{1,\dots,n\} | \{v_j, v_\jmath\} \in E} \mathcal{L}^{l+1}_a = \jmath$
  for every $a \in A, j \in \{1,2, \dots, n\}$ and $l \in \{0,1, \dots, \eta - 1\}$
  (agent a moves along edges only or stay in a vertex)
- $\bigwedge_{b \in A | b \neq a} \mathcal{L}^{l+1}_a \neq \mathcal{L}^l_b$ for every $a \in A$ and $l \in \{0,1, \dots, \eta - 1\}$ (20)

(target vertex of agent's a move must be empty)

- AllDifferent$(\mathcal{L}_{a_1}^l, \mathcal{L}_{a_2}^l, \dots, \mathcal{L}_{a_\mu}^l)$ for every $l \in \{0,1,\dots,\eta\}$ (21)
  (at most one agent reside in each vertex at each time step). □

Initial and goal arrangements will be expressed though the following constraints:

- $\mathcal{L}_a^0 = j$ for $a \in A$ with $\alpha_0(a) = v_j$ } Initial locations (22)
- $\mathcal{L}_a^\eta = j$ for $a \in A$ with $\alpha_+(a) = v_j$ } Goal locations (23)

Again, finite domain state variables $\mathcal{L}_a^l$ are represented as a bit vector (vector of propositional variables) using binary encoding. That is, $\lceil \log_2 |V| \rceil$ propositional variables are introduced for each $\mathcal{L}_a^l$ variable. The resulting formula in CNF will be denoted as $F_{DIFF}(\eta, \Sigma)$.

AllDifferent$(\mathcal{L}_{a_1}^l, \mathcal{L}_{a_2}^l, \dots, \mathcal{L}_{a_\mu}^l)$ constraint requires that all the involved variables are assigned different values; that is, $\bigwedge_{j,k \in \{1,2,\dots,\mu\} | j<k} \mathcal{L}_{a_j}^l \neq \mathcal{L}_{a_k}^l$. Differences between finite domain state variables are encoded using the scheme introduced in [1]. The scheme is used to encode constraints (20) as well as (21). Inequality between variables $\mathcal{L}_{a_j}^l$ and $\mathcal{L}_{a_k}^l$ is expressed in the scheme by the following clauses. Auxiliary variables $d_{j,k}^l$ representing difference at individual bits are introduced.

$$\text{var}_{\neq}\left(\mathcal{L}_{a_j}^l, \mathcal{L}_{a_k}^l\right) = \bigvee_{\mathbb{i}=0}^{\lceil \log_2 n \rceil - 1} d_{j,k}^l \tag{24}$$

$$\text{where} \quad \bigwedge_{\mathbb{i}=0}^{\lceil \log_2 n \rceil - 1} \left(\neg d_{j,k}^l \vee \neg \mathcal{L}_{a_j}^l[\mathbb{i}] \vee \mathcal{L}_{a_k}^l[\mathbb{i}]\right) \wedge \left(\neg d_{j,k}^l \vee \mathcal{L}_{a_j}^l[\mathbb{i}] \vee \neg \mathcal{L}_{a_k}^l[\mathbb{i}]\right)$$

Conditional equality disjunction (19) is encoded by introducing auxiliary propositional variables to represent equalities between bit vectors. For each $j \in \{1,2,\dots,n\}$ (that is, for each vertex), agent $a \in A$, and time layer $l \in \{0,1,\dots,\eta\}$, an auxiliary variable $e_{a,j}^l$ which stands for equality $\mathcal{L}_a^l = j$ is introduced. The link between auxiliary variables $e_{a,j}^l$ and actual equalities is established through the following constraint:

$$e_{a,j}^l \Leftrightarrow \text{con}_{=}(\mathcal{L}_a^l, j) \tag{25}$$

Then moving along edges – constraints (19) – can be easily expressed as single clause over auxiliary variables:

$$e_{a,j}^{l} \Rightarrow \neg e_{a,j}^{l+1} \vee \bigvee_{\mathit{j} \in \{1,\ldots,n\} | \{v_j, v_{\mathit{j}}\} \in E} e_{a,\mathit{j}}^{l+1} \tag{26}$$

Again, the space consumption of the ALL-DIFFERENT encoding will be calculated for regular time layers only.

Proposition 4 ***(ALL-DIFFERENT ENCODING SIZE).*** *The number of visible propositional variables in* $F_{DIFF}(\eta, \Sigma)$ *is* $\mathcal{O}(\eta \cdot \mu \cdot \lceil \log_2 |V| \rceil)$ *and there are* $\mathcal{O}(\eta \cdot \mu \cdot |V|)$ *auxiliary variables; that is, the number of variables is* $\mathcal{O}(\eta \cdot \mu \cdot |V|)$. *The number of clauses is* $\mathcal{O}\left(\eta \cdot \lceil \log_2 |V| \rceil \cdot \left(\binom{\mu}{2} + \mu \cdot |V|\right)\right)$. ■

Proof. Let us calculate the number of variables and clauses. Each variable $\mathcal{L}_a^l$ is represented by $\log_2 |V|$ variables and the number of $\mathcal{L}_a^l$ variables is $(\eta + 1) \cdot \mu$. For each $\mathcal{L}_a^l$ variable and its value, an auxiliary variable is introduced. As $\mathcal{L}_a^l$ can take $|V|$ values, we get the result that there are $(\eta + 1) \cdot \mu \cdot (\log_2 |V| + |V|)$ variables which is $\mathcal{O}(\eta \cdot \mu \cdot |V|)$.

A single time layer requires as many as $\binom{\mu}{2}$ inequalities between all pairs of $\mathcal{L}_a^l$ variables corresponding to distinct agents to model the AllDifferent constraint from (21). Each inequality is modeled by $2 \cdot \lceil \log_2 |V| \rceil$ ternary clauses plus one clause of arity $\lceil \log_2 |V| \rceil$. This is in total $(\eta + 1) \cdot \binom{\mu}{2} \cdot (2 \cdot \lceil \log_2 |V| \rceil + 1)$ clauses.

Next, we need as many as $\binom{\mu}{2}$ inequalities between $\mathcal{L}_a^l$ variables from two consecutive time layers (constraint (20)) which adds the same number of $(\eta + 1) \cdot \binom{\mu}{2} \cdot (2 \cdot \lceil \log_2 |V| \rceil + 1)$ clauses again.

Links between auxiliary variables $e_{a,j}^{l}$ and actual equalities (25) they represent need $\lceil \log_2 |V| \rceil$ binary clauses plus one clause of arity $\lceil \log_2 |V| \rceil + 1$, which is $(\eta + 1) \cdot \mu \cdot |V| \cdot (\lceil \log_2 |V| \rceil + 1)$ in total.

Finally, constraints expressing that agents move along edges only (26) contribute to each vertex v_j in $\text{Exp}_{\text{T}}(G, \eta)$ at given time layer except the last one by μ clauses of arity $\deg_{\text{G}}(v_j) + 2$ which is $\eta \cdot \mu \cdot |V|$ clauses in total.

Altogether we have $(\eta + 1) \cdot (\binom{\mu}{2} \cdot (2 \cdot \lceil \log_2 |V| \rceil + 1) + \mu \cdot |V| \cdot (\lceil \log_2 |V| \rceil + 1)) + \eta \cdot \mu \cdot |V|$ clauses in $F_{DIFF}(\eta, \Sigma)$ encoding which is $\mathcal{O}(\eta \cdot \lceil \log_2 |V| \rceil \cdot \left(\binom{\mu}{2} + \mu \cdot |V|\right))$. ■

Proposition 5 ***(PATHS AND*** $F_{DIFF}(\eta,\Sigma)$ ***SATISFACTION).*** *A set* $\Pi = \{\pi_1, \pi_2, \ldots, \pi_\mu\}$ *of non-overlapping vertex disjoint paths in* $\text{Exp}_\text{T}(G,\eta)$ *so that* π_i *connects* $[\alpha_0(a_i),0]$ *with* $[\alpha_+(a_i),\eta]$ *for* $i = 1,2,\ldots,\mu$ *exists if and only if* $F_{DIFF}(\eta,\Sigma)$ *is satisfiable. Moreover, paths* $\pi_1, \pi_2, \ldots, \pi_\mu$ *can be unambiguously constructed from satisfying valuation of* $F_{DIFF}(\eta,\Sigma)$ *and vice versa.* ■

Sketch of proof. For simplicity, we will work on the level of finite domain state variables. Assume that non-overlapping vertex disjoint paths $\pi_1, \pi_2, \ldots, \pi_\mu$ exist in $\text{Exp}_\text{T}(G,\eta)$. The satisfying valuation of $F_{DIFF}(\eta,\Sigma)$ can be directly constructed from these paths. Let $\pi_i = ([u_0,0], [u_1,1], [u_2,2], \ldots, [u_\eta,\eta])$, $u_l \in V$ for $l = 0,1,\ldots,\eta$ where $u_0 = \alpha_0(a_i)$ and $u_\eta = \alpha_+(a_i)$. Then finite domain state variables will be set as follows: $\mathcal{L}_{a_i}^0 = u_0$, $\mathcal{L}_{a_i}^1 = u_1$, …, $\mathcal{L}_{a_i}^\eta = u_\eta$ for every $i = 1,2,\ldots,\mu$. The assumptions that paths were vertex disjoint and non-overlapping ensure that constraints (21) and (20) respectively are satisfied. Consecutive vertices in paths are connected by directed edges that correspond to edges in G. Hence, constraints (19) are satisfied.

Assume on the other hand that we have a satisfying valuation of $F_{DIFF}(\eta,\Sigma)$. We can immediately set $\pi_i = ([\mathcal{L}_{a_i}^0,0], [\mathcal{L}_{a_i}^1,1], [\mathcal{L}_{a_i}^2,2], \ldots, [\mathcal{L}_{a_i}^\eta,\eta])$ for every $i = 1,2,\ldots,\mu$. Satisfaction of constraints (19) ensures that constructed sequences of vertices are paths in $\text{Exp}_\text{T}(G,\eta)$ which are moreover vertex disjoint and non-overlapping due to constraints (21) and (20). ■

CPF solving via $F_{DIFF}(\eta,\Sigma)$ satisfaction is justified by the following theorem which can be shown by combining Proposition 1 and just proven Proposition 5 (from $F_{DIFF}(\eta,\Sigma)$ satisfaction non-overlapping vertex disjoint paths can be obtained which correspond to CPF solution).

Theorem 2 ***(SOLUTION OF*** Σ ***AND*** $F_{DIFF}(\eta,\Sigma)$ ***SATISFACTION).*** *A solution of a CPF* $\Sigma = (G, A, \alpha_0, \alpha_+)$ *exists if and only if there exist* $\eta \in \mathbb{N}$ *for that formula* $F_{DIFF}(\eta,\Sigma)$ *is satisfiable.* ■

4.2.3. MATCHING Propositional Encoding

We observed that vertex disjoint non-overlapping paths in time expansion graph resemble a *commodity flow* [1] in a network of time expansion graph where vertices and edges are assigned unit capacities. The intuition is that edges included into paths should be saturated by one unit of the flow. Such setting conveys commodity from each initial vertex to each goal vertex. However, the correspondence between paths of required properties and flow works in one direction only. The

flow reflects well the requirement that paths should be vertex disjoint but does not simulate non-overlapping between paths as well as the correct interconnection between initial and goal vertex of the same agents (the flow may interconnect initial and goal vertices of two distinct agents).

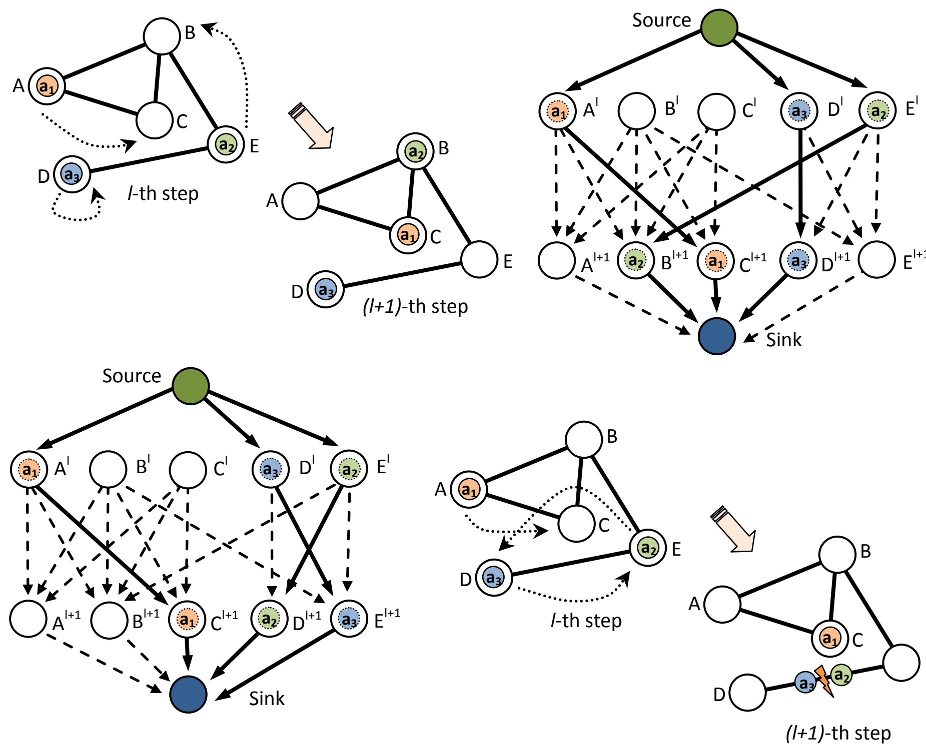


Figure 4. *Correspondence of agent movement and flow in bipartite graph.* Movement between time steps l and $l+1$ and corresponding flow in bipartite graph made of l-th and $(l+1)$-th time layer where vertices and edges are assigned unit capacities is shown. Valid movement always induces flow in which saturated edges are non-overlapping; that is, $\{A, E\} \cap \{B, C\} = \emptyset$ (*upper part*). On the other hand, flow does not necessarily induce non-overlapping edges which may result in invalid movement (*lower part*).

The design of the MATCHING encoding will follow the intuition suggested by single commodity flows. It will be divided into two parts – the first part, called FLOW part, will check the existence of a flow that generates non-overlapping vertex disjoint paths. This part can be regarded as an encoding of a relaxed CPF with anonymous agents where we care about relocation of a group of agents to a set of goal vertices but we don't care about what particular agent arrives at particular goal vertex (generated paths may interconnect initial and goal vertices of distinct agents). The second part, called MAPPING part, of the encoding maps

distinguishable agents to paths marked out by the flow, which eventually override the relaxation from the first part of the encoding. The encoding should allow fast testing of the existence of non-overlapping flow to enable using this test as a heuristic since its existence is necessary condition for existence of a solution.

Definition 7 (MATCHING ENCODING – $F_{\eta-MATCH}^{FLOW}(\Sigma)$). A *FLOW part of the MATCHING encoding* of given CPF $\Sigma = [G, A, \alpha_0, \alpha_+]$ with $G = (V, E)$ consists of a propositional variable for each vertex and edge in the time expansion graph that model its saturation by the flow. That is, propositional variable $\mathcal{M}_v^l$ is introduced for every $l = 0,1,\ldots,\eta$ and $v \in V$ and propositional variables $\mathcal{E}_{u,v}^l$ and $\mathcal{E}_u^l$ are introduced for every $l = 0,1,\ldots,\eta$ and $\{u,v\} \in E$ and $u \in V$ respectively. Constraints enforce that variables set to $TRUE$ form a non-overlapping flow:

- $\mathcal{E}_{u,v}^l \Rightarrow \mathcal{M}_u^l \wedge \mathcal{M}_v^{l+1}$ for every $\{u,v\} \in E$ and $l \in \{0,1,\ldots,\eta-1\}$, (25)
  $\mathcal{E}_u^l \Rightarrow \mathcal{M}_u^l \wedge \mathcal{M}_u^{l+1}$ for every $u \in V$ and $l \in \{0,1,\ldots,\eta-1\}$
  (if an edge is selected into flow then its endpoints are selected as well)
- $\mathcal{E}_u^l + \sum_{v|\{u,v\}\in E} \mathcal{E}_{u,v}^l \leq 1$ for every $u \in V$ and $l \in \{0,1,\ldots,\eta-1\}$, (26)
  $\mathcal{E}_v^l + \sum_{u|\{u,v\}\in E} \mathcal{E}_{u,v}^l \leq 1$ for every $v \in V$ and $l \in \{0,1,\ldots,\eta-1\}$,
  (at most one incoming and outgoing edge is selected into flow)
- $\mathcal{M}_u^l \Rightarrow \mathcal{E}_u^l \vee \bigvee_{v|\{u,v\}\in E} \mathcal{E}_{u,v}^l$ for every $u \in V$ and $l \in \{0,1,\ldots,\eta-1\}$, (27)
  $\mathcal{M}_v^{l+1} \Rightarrow \mathcal{E}_u^l \vee \bigvee_{u|\{u,v\}\in E} \mathcal{E}_{u,v}^l$ for every $v \in V$ and $l \in \{0,1,\ldots,\eta-1\}$,
  (if a vertex is selected into flow then at least one outgoing and incoming edge must be selected as well)
- $\mathcal{E}_{u,v}^l \Rightarrow \neg\mathcal{M}_v^l$ for every $\{u,v\} \in E$ and $l \in \{0,1,\ldots,\eta-1\}$, (28)
  (source and target vertices of non-trivial moves must be disjoint). □

The second part of the encoding where individual distinguishable agents manifest themselves is introduced in the following definition.

Definition 8 (MATCHING ENCODING – $F_{\eta-MATCH}^{MAP}(\Sigma)$). A *MAPPING part of the MATCHING encoding* of given CPF $\Sigma = [G, A, \alpha_0, \alpha_+]$ with $G = (V, E)$ consists of a finite domain variable $\mathcal{A}_v^l \in \{0,1,\ldots,\mu\}$ for each vertex $v \in V$ and every time layer $l = 0,1,\ldots,\eta$ to model agent occurrence in a vertex. Constraints interconnect the MAPPING part with FLOW part so that actual agents follow paths indicated by the flow:

- $\mathcal{E}_{u,v}^{l} \Rightarrow \mathcal{A}_u^l = \mathcal{A}_v^{l+1}$ for every $\{u, v\} \in E$ and $l \in \{0,1, \dots, \eta - 1\}$, (29)
  (<u>if an edge is saturated by the flow then the same agent appears at its both ends</u>)
  (30)
- $\mathcal{A}_u^l \neq 0 \Rightarrow \mathcal{M}_u^l$ for every $u \in V$ and $l \in \{0,1, \dots, \eta\}$
  (<u>if an agent occurs in a vertex then the vertex is saturated by the flow</u>) □

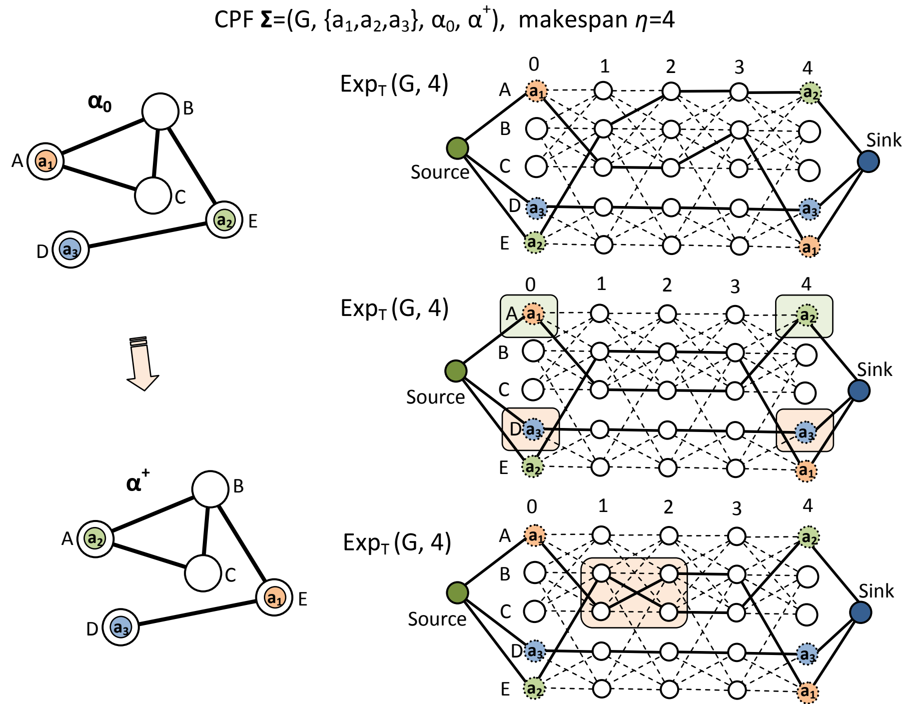


Figure 5. *Non-overlapping vertex disjoint paths in time expansion graph depicted and single commodity flow correspondence.* Edges and vertices in time expansion graph are assumed to have unit capacities. A correct flow can be reconstructed from vertex disjoint non-overlapping paths (*upper right part*). On the other hand, flow does not necessarily correspond to paths of required properties (*middle right part* shows connection of initial and goal vertices of different agents and lower right part shows overlapping paths between time layers 1 and 2).

As in the previous encodings $\mathcal{A}_v^l$ variables having $\mu + 1$ states are represented by $\lceil \log_2(\mu + 1) \rceil$ propositional variables using binary encoding. Initial and goal arrangements will be expressed though the following constraints:

- $\mathcal{A}_u^0 = i \wedge \mathcal{M}_u^0$ for $u \in V$ if there is $i \in \{1,2,\dots,\mu\}$ such that $\alpha_0(a_i) = u$ (31)
- $\mathcal{A}_u^0 = 0 \wedge \neg\mathcal{M}_u^0$ for $u \in V$ if $(\forall a \in A)\alpha_0(a) \neq u$ (32)

  Initial locations (31–32)
- $\mathcal{A}_u^\eta = i \wedge \mathcal{M}_u^\eta$ for $u \in V$ if there is $i \in \{1,2,\dots,\mu\}$ such that $\alpha_+(a_i) = u$ (33)
- $\mathcal{A}_u^\eta = 0 \wedge \neg\mathcal{M}_u^\eta$ for $u \in V$ if $(\forall a \in A)\alpha_+(a) \neq u$ (34)

  Goal locations (33–34)

The resulting formula of the MATCHING encoding in CNF is a conjunction of the FLOW part, MAPPING part, and boundary conditions and is denoted as $F_{\eta-MATCH}(\Sigma)$. To obtain CNF it is necessary to rewrite (26) as clauses. That is, for example $\mathcal{E}_u^l + \sum_{v|\{u,v\}\in E} \mathcal{E}_{u,v}^l \leq 1$ is rewritten as a conjunction of clauses that forbid all pairs of involved variables to be set to $TRUE$ simultaneously:

$$\begin{gathered}\bigwedge\nolimits_{v,w|\{u,v\}\in E \wedge \{u,w\}\in E \wedge v\neq w} \neg\mathcal{E}_{u,v}^l \vee \neg\mathcal{E}_{u,w}^l \\ \bigwedge\nolimits_{v|\{u,v\}\in E} \neg\mathcal{E}_{u,v}^l \vee \neg\mathcal{E}_u^l\end{gathered} \tag{35}$$

Binary encoded variables $\mathcal{A}_v^l$ are not involved in any complex relation – only conditional equality between these variables are introduced while all other modeling issues concerning validity conditions are done in the FLOW part of the encoding.

The conditional equality between $\mathcal{A}_u^l$ and $\mathcal{A}_v^{l+1}$ (25) can be expressed using construct introduced earlier:

$$\mathcal{E}_{u,v}^l \Rightarrow \mathrm{var}_=(\mathcal{A}_u^l, \mathcal{A}_v^{l+1}) \tag{36}$$

Constraint (26) can be also easily rewritten as follows:

$$\bigwedge\nolimits_{\mathbb{i}=0}^{\lceil\log_2(\mu+1)\rceil-1} \neg\mathcal{A}_u^l[\mathbb{i}] \vee \mathcal{M}_u^l \tag{37}$$

Proposition 6 ***(MATCHING ENCODING SIZE).*** *The number of propositional variables in* $F_{\eta-MATCH}(\Sigma)$ *is* $\mathcal{O}(\eta \cdot (|E| + |V| \cdot \lceil\log_2(\mu)\rceil))$. *The number of clauses is* $\mathcal{O}\left(\eta \cdot \left((|V| + |E|) \cdot \lceil\log_2(\mu)\rceil + \sum_{v\in V}\binom{\deg_G(v)}{2}\right)\right)$. ■

Proof. The FLOW part of the MATCHING encoding has a propositional variable $\mathcal{M}_v^l$ for each vertex $v \in V$ and time layer $l \in \{0,1,\dots,\eta\}$ and $\mathcal{E}_{u,v}^l$ for each edge

$\{u,v\} \in E$ and time layer, which in total makes $(\eta+1)\cdot(|V|+|E|)$ propositional variables. Further, we have a vector of $\lceil \log_2(\mu+1)\rceil$ propositional variables representing $\mathcal{A}_v^l$ for each vertex and time layer in the MAPPING part. This in total makes another $(\eta+1)\cdot|V|\cdot\lceil\log_2(\mu+1)\rceil$ variables. Altogether, there are $(\eta+1)\cdot(|E|+|V|+|V|\cdot\lceil\log_2(\mu+1)\rceil)$ variables which is $\mathcal{O}\big(\eta\cdot(|E|+|V|\cdot\lceil\log_2(\mu)\rceil)\big)$.

Constraints (25) develops into $\eta\cdot(|V|+|E|)$ ternary clauses. Constraints (26) develop into $2\cdot\eta\cdot\sum_{v\in V}\binom{\deg_G(v)+1}{2}$ binary clauses as indicated by (31). Constraints (27) introduce two clauses of length $\deg_G(v)+1$ for each vertex and time layer; that is, $2\cdot\eta\cdot|V|$ clauses are added. Finally, constraints (28) add a binary clause for each vertex and time layer, which is again dominated by previous expressions. Conditional equality between two bit vectors in (29) develops into $2\cdot\lceil\log_2(\mu+1)\rceil$ ternary clauses while the equality is introduced for each edge and time layer; that is, $2\cdot\eta\cdot|E|\cdot\lceil\log_2(\mu+1)\rceil$ ternary clauses are added. It is easy to observe that expression (30) represents $(\eta+1)\cdot|E|\cdot\lceil\log_2(\mu+1)\rceil$ binary clauses. Altogether, there are $\eta\cdot(3\cdot|V|+|E|+2\cdot\sum_{v\in V}\binom{\deg_G(v)+1}{2}+2\cdot|E|\cdot\lceil\log_2(\mu+1)\rceil)+(\eta+1)\cdot|E|\cdot\lceil\log_2(\mu+1)\rceil$ clauses which is $\mathcal{O}\left(\eta\cdot\left((|V|+|E|)\cdot\lceil\log_2(\mu)\rceil+\sum_{v\in V}\binom{\deg_G(v)}{2}\right)\right)$. ■

Proposition 7 ***(PATHS AND*** $F_{MATCH}(\eta,\Sigma)$ ***SATISFACTION).*** *A set* $\Pi=\{\pi_1,\pi_2,\ldots,\pi_\mu\}$ *of non-overlapping vertex disjoint paths in* $\mathrm{Exp_T}(G,\eta)$ *so that* π_i *connects* $[\alpha_0(a_i),0]$ *with* $[\alpha_+(a_i),\eta]$ *for* $i=1,2,\ldots,\mu$ *exists if and only if* $F_{MATCH}(\eta,\Sigma)$ *is satisfiable. Moreover, paths* $\pi_1,\pi_2,\ldots,\pi_\mu$ *can be unambiguously constructed from satisfying valuation of* $F_{MATCH}(\eta,\Sigma)$ *and vice versa.* ■

Sketch of proof. We will work at the level of finite domain state variables $\mathcal{A}_v^l$ instead of bit vectors to simplify the proof.

Assume that non-overlapping vertex disjoint paths $\pi_1,\pi_2,\ldots,\pi_\mu$ exist so that π_i connects $[\alpha_0(a_i),0]$ with $[\alpha_+(a_i),\eta]$ in $\mathrm{Exp_T}(G,\eta)$. Let $\pi_i=([u_0,0],[u_1,1],[u_2,2],\ldots,[u_\eta,\eta])$, with $u_l\in V$ for $l=0,1,\ldots,\eta$ where $u_0=\alpha_0(a_i)$ and $u_\eta=\alpha_+(a_i)$. The satisfying valuation of $F_{MATCH}(\eta,\Sigma)$ can be easily constructed by setting $\mathcal{A}_{u_0}^0=i$, $\mathcal{A}_{u_1}^1=i$, ..., $\mathcal{A}_{u_\eta}^\eta=i$. Next, variables representing flow $\mathcal{M}_{u_0}^0$, $\mathcal{M}_{u_1}^1$, ..., $\mathcal{M}_{u_\eta}^\eta$ are set to $TRUE$ and $\mathcal{E}_{u_0,u_1}^0$, $\mathcal{E}_{u_1,u_2}^1$, ..., $\mathcal{E}_{u_{\eta-1},u_\eta}^{\eta-1}$ are set to $TRUE$ as well (the convention that $\mathcal{E}_{u_l,u_{l+1}}^l\equiv\mathcal{E}_{u_l}^l$ if $u_l=u_{l+1}$ is used here). Now, observe that all the constraints are satisfied. The interconnection between the FLOW and the MAPPING part (constraints (25) and (26)) is satisfied by the construction so we just need to check constraints in the FLOW part of the encoding. Propagation

of the flow from edges to vertices (constraints (24)) is also ensured by the construction. The fact that original paths are vertex disjoint ensures validity of constraints (25) and (26) which together enforce selection of exactly one incoming and one outgoing edge through setting $\mathcal{E}_{u,v}^{l}$ variables for each vertex saturated by the flow indicated by $\mathcal{M}_{v}^{l}$ variable set to $TRUE$. Finally, the non-overlapping property of paths is directly translated to satisfaction of constraints (28). Initial and goal location constraints are trivially satisfied. Altogether, $F_{MATCH}(\eta, \Sigma)$ is satisfied by constructed valuation of its variables.

Now let us check the opposite implication. Assume that $F_{MATCH}(\eta, \Sigma)$ is satisfiable. Let $\pi_i = ([u_0, 0], [u_1, 1], [u_2, 2], \ldots, [u_\eta, \eta])$ such that $u_0 = \alpha_0(a_i)$ and $\mathcal{E}_{u_l, u_{l+1}}^{l}$ is $TRUE$ for each $l = 0,1, \ldots, \eta - 1$. This can be done due to constraints (25) - (27) that propagate flow from the initial positions in the first time layer towards final layer. We shall verify that paths constructed in this way have required properties – are vertex disjoint non-overlapping and interconnects initial and goal positions of agents. FLOW part of the encoding ensures that constructed paths are vertex disjoint and non-overlapping. We need just to add non-overlapping to already checked flow propagation. The non-overlapping is established by constraints (28). However, the FLOW part does not ensure that $u_\eta = \alpha_+(a_i)$; satisfaction of the FLOW part alone may result in a path that interconnects initial and goal positions of two distinct agents. This is corrected by constraints included in the MAPPING part of the encoding. These constraints propagate agent a_i along edges $\{u_l, u_{l+1}\}$ and eventually force it to appear in u_η where goal constraints (33) and (34) ensure that agent a_i arrives to the right vertex. ■

By combining just proven Proposition 7 and correspondence between non-overlapping vertex disjoint paths in $\text{Exp}_\text{T}(G, \eta)$ the following theorem can be immediately obtained.

Theorem 3 ***(SOLUTION OF*** Σ ***AND*** $F_{MATCH}(\eta, \Sigma)$ ***SATISFACTION).*** *A solution of a CPF* $\Sigma = (G, A, \alpha_0, \alpha_+)$ *exists if and only if there exist* $\eta \in \mathbb{N}$ *for that formula* $F_{MATCH}(\eta, \Sigma)$ *is satisfiable.* ■

4.2.4. DIRECT/SIMPLIFIED Propositional Encoding

As all the previous encodings of CPF used binary representation of agent occurrence in a vertex in some form, we also considered encoding of CPF that expresses agent occurrences in vertices directly. That is, there will be single propositional variable that encodes occurrence of a given agent in a given vertex at a given time-step. The resulting CPF encoding will be called DIRECT. While the

design of variables is extremely simple in the DIRECT encoding, the set of constraints is more complex as summarized in the following definition [30].

Definition 9 (DIRECT ENCODING – $F_{DIR}(\eta, \Sigma)$). Assume that a CPF $\Sigma = [G, A, \alpha_0, \alpha_+]$ with $G = (V, E)$ is given. A *DIRECT encoding* for CPF Σ consists of propositional variables $\mathcal{X}_{a,v}^{l}$ for every $a \in A$, $v \in V$, and time layer $l \in \{0,1,\ldots,\eta\}$ to model occurrences of agents in vertices over time. The interpretation is that $\mathcal{X}_{a,v}^{l}$ is assigned $TRUE$ if and agent a appears in vertex v at time step l. The following constrains ensure satisfaction of validity conditions between consecutive arrangements of agents:

- $\bigwedge_{u,v \in V, u \neq v} \neg\mathcal{X}_{a,u}^{l} \vee \neg\mathcal{X}_{a,v}^{l}$ for every $l \in \{0,1,\ldots,\eta\}$ (38)
  $\bigvee_{v \in V} \mathcal{X}_{a,v}^{l}$ and $a \in A$
  (an agent is placed in exactly one vertex at each time step)
- $\bigwedge_{a,b \in A, a \neq b} \neg\mathcal{X}_{a,v}^{l} \vee \neg\mathcal{X}_{b,v}^{l}$ for every $l \in \{0,1,\ldots,\eta\}$ (39)
  and $v \in V$
  (at most one agent is placed in each vertex at each time step)
- $\mathcal{X}_{a,v}^{l} \Rightarrow \mathcal{X}_{a,v}^{l+1} \vee \bigvee_{u \in V, \{v,u\} \in E} \mathcal{X}_{a,u}^{l+1}$ for every $l \in \{0,1,\ldots,\eta-1\}$,   (40)
  $\mathcal{X}_{a,v}^{l+1} \Rightarrow \mathcal{X}_{a,v}^{l} \vee \bigvee_{u \in V, \{v,u\} \in E} \mathcal{X}_{a,u}^{l}$ $v \in V$, and $a \in A$
  (an agent relocates to some of its neighbors or makes no move)
- $\mathcal{X}_{a,v}^{l} \wedge \mathcal{X}_{a,u}^{l+1} \Rightarrow \bigwedge_{b \in A} \neg\mathcal{X}_{b,u}^{l} \wedge \bigwedge_{b \in A} \neg\mathcal{X}_{b,v}^{l+1}$ (41)
  for every $l \in \{0,1,\ldots,\eta-1\}$, $u, v \in V$
  such that $\{u,v\} \in E$ and $a \in A$
  (target vertex of a move must be vacant and the source vertex will be vacant after the move is performed). □

Initial and goal arrangements will be expressed though the following constraints:

- $\mathcal{X}_{a,v}^{0}$ for $v \in V$ if there is $a \in A$ such that $\alpha_0(a) = v$ (42)
- $\neg\mathcal{X}_{a,v}^{0}$ otherwise   (43)

  } Initial locations
- $\mathcal{X}_{a,v}^{\eta}$ for $v \in V$ if there is $a \in A$ such that $\alpha_+(a) = v$ (44)
- $\neg\mathcal{X}_{a,v}^{\eta}$ otherwise   (45)

  } Goal locations

The resulting DIRECT encoding formula in CNF will be denoted as $F_{DIR}(\eta, \Sigma)$. It can be easily observed that the vacancy of target vertex and source

vertex before and after the move (constraint (41)) is quite repetitive as the right side of the implication is independent of agent a. Therefore, the encoding is enhanced by introducing auxiliary variables $\mathcal{E}_u^l$ for each vertex $u \in V$ and time layer $l \in \{0,1,\dots,\eta\}$ that represent vacancy of vertex u at time step l. Semantics of $\mathcal{E}_u^l$ variables is represented by the following constraint:

- $\mathcal{E}_u^l \Rightarrow \bigwedge_{a \in A} \neg \mathcal{X}_{a,u}^l$ for every $l \in \{0,1,\dots,\eta\}$ and $u \in V$ (46)
  (<u>in an empty vertex no agent can appear at given time</u>)

The repetitive part in constraint (41) can be then replaced by its version with auxiliary variables as follows:

- $\mathcal{X}_{a,v}^l \wedge \mathcal{X}_{a,u}^{l+1} \Rightarrow \mathcal{E}_u^l \wedge \mathcal{E}_v^{l+1}$ for every $l \in \{0,1,\dots,\eta-1\}$, $u,v \in V$ such that $\{u,v\} \in E$ and $a \in A$. (47)

The resulting encoding with auxiliary variables will be called SIMPLIFIED and the corresponding CNF formula will be denoted as $F_{SIM}(\eta,\Sigma)$.

Proposition 8 ***(DIRECT/SIMPLIFIED ENCODING SIZE).*** *The number of propositional variables in $F_{DIR}(\eta,\Sigma)$ is $\mathcal{O}(\eta \cdot \mu \cdot |V|)$. The number of clauses is $\mathcal{O}\big(\eta \cdot (\mu \cdot |V|^2 + \mu^2 \cdot |V| + \mu^2 \cdot |E|)\big)$. $F_{SIM}(\eta,\Sigma)$ contains additional $\mathcal{O}(\eta \cdot |V|)$ propositional variables while the total number of clauses is $\mathcal{O}\big(\eta \cdot (\mu \cdot |V|^2 + \mu^2 \cdot |V| + \mu \cdot |E|)\big)$.* ■

Proof. It is easy to see that there are exactly $(\eta+1) \cdot \mu \cdot |V|$ variables $\mathcal{X}_{a,v}^l$ and $\eta \cdot |V|$ $\mathcal{E}_u^l$ variables just by calculating their index scopes which gives us the result regarding the number of propositional variables in $F_{DIR}(\eta,\Sigma)$ and $F_{SIM}(\eta,\Sigma)$.

Every time layer and agent adds $\binom{|V|}{2}$ binary clauses and one $|V|$-ary clause within constraints (38). Thus, we have $(\eta+1) \cdot \mu \cdot \binom{|V|}{2}$ binary clauses and $(\eta+1) \cdot \mu$ $|V|$-ary from writing constraints (38) as clauses in total. The similar calculation can be done for constraints (39); we have $\binom{\mu}{2}$ binary clauses for each time layer and a vertex; that is, $(\eta+1) \cdot |V| \cdot \binom{\mu}{2}$ binary clauses in total.

There are two $(\deg_G(v)+2)$-ary clauses for every vertex v in every time layer except the last one and for every agent from constraints (40), which in total gives $\eta \cdot \mu$ $(\deg_G(v)+2)$-ary clauses for each vertex $v \in V$. That is, $\eta \cdot \mu \cdot |V|$ clauses in total.

Note that each implication in constraint (41) develops into $2 \cdot \mu$ ternary clauses. There are $|E|$ such groups of clauses for every agent and a time layer except

the last one. Thus, $2 \cdot \eta \cdot \mu^2 \cdot |E|$ ternary clauses are needed in total for expressing constraints (41).

Altogether, the total number of clauses in $F_{DIR}(\eta,\Sigma)$ is $(\eta+1) \cdot \left(\mu \cdot \left(\binom{|V|}{2}+1\right) + |V| \cdot \binom{\mu}{2}\right) + 2 \cdot \eta \cdot \mu \cdot (|V| + \mu \cdot |E|)$ which is $\mathcal{O}\big(\eta \cdot (\mu \cdot |V|^2 + \mu^2 \cdot |V| + \mu^2 \cdot |E|)\big)$.

Constraints (47) develop into smaller number of clauses if compared with the original constraints (41) which they replace because of the shorter right hand side in the implication in $F_{SIM}(\eta,\Sigma)$. Concretely, each implication from (47) develops into exactly 2 ternary clauses which gives $2 \cdot \eta \cdot \mu \cdot |E|$ ternary clauses in total.

Interconnection of auxiliary variables $\mathcal{E}_u^l$ with their meaning requires μ binary clauses per one implication from constraint (46). There are as many as $(\eta+1) \cdot |V|$ such interconnections which results in $(\eta+1) \cdot \mu \cdot |V|$ in total. Hence, the total number of clauses in $F_{SIM}(\eta,\Sigma)$ is $(\eta+1) \cdot \left(\mu \cdot \left(\binom{|V|}{2}+1\right) + |V| \cdot \binom{\mu}{2}\right) + \eta \cdot \mu \cdot (|V| + 2 \cdot |E|)$ which is $\mathcal{O}\big(\eta \cdot (\mu \cdot |V|^2 + \mu^2 \cdot |V| + \mu \cdot |E|)\big)$. ■

Proposition 9 ***(PATHS AND*** $F_{DIR}(\eta,\Sigma)/F_{SIM}(\eta,\Sigma)$ ***SATISFACTION).*** *A set* $\Pi = \{\pi_1, \pi_2, \ldots, \pi_\mu\}$ *of non-overlapping vertex disjoint paths in* $\mathrm{Exp_T}(G,\eta)$ *so that* π_i *connects* $[\alpha_0(a_i), 0]$ *with* $[\alpha_+(a_i), \eta]$ *for* $i = 1,2,\ldots,\mu$ *exists if and only if* $F_{DIR}(\eta,\Sigma)$ *is satisfiable. Moreover, paths* $\pi_1, \pi_2, \ldots, \pi_\mu$ *can be unambiguously constructed from satisfying valuation of* $F_{DIR}(\eta,\Sigma)$ *and vice versa. The same hold for* $F_{SIM}(\eta,\Sigma)$. ■

Sketch of proof. Assume that non-overlapping vertex disjoint paths $\pi_1, \pi_2, \ldots, \pi_\mu$ exist so that π_i connects $[\alpha_0(a_i), 0]$ with $[\alpha_+(a_i), \eta]$ in $\mathrm{Exp_T}(G,\eta)$. We will construct a satisfying valuation of $F_{DIR}(\eta,\Sigma)$ from $\pi_1, \pi_2, \ldots, \pi_\mu$.

Let $\pi_i = ([u_0, 0], [u_1, 1], [u_2, 2], \ldots, [u_\eta, \eta])$, with $u_l \in V$ for $l = 0,1,\ldots,\eta$ where $u_0 = \alpha_0(a_i)$ and $u_\eta = \alpha_+(a_i)$, then variables $\mathcal{X}_{a_i,u_0}^0$, $\mathcal{X}_{a_i,u_1}^1$, ..., $\mathcal{X}_{a_i,u_\eta}^\eta$ will be set to $TRUE$. This setup of $\mathcal{X}_{a,v}^l$ variables will be set for every $i = 1,2,\ldots,\mu$.

It is now easy to verify that all the constraints from the DIRECT encoding hold. Constraints (38) hold because each directed path π_i intersects the time layer in exactly one vertex. Constraints (39) hold since directed paths are vertex disjoint. As paths go from one time layer to the next, constraints (40) hold as well. Finally, since paths are non-overlapping constraints (41) also hold.

Satisfying valuation of $F_{SIM}(\eta,\Sigma)$ requires assigning truth values to $\mathcal{E}_u^l$ variables in addition. Nevertheless, truth values of $\mathcal{E}_u^l$ are directly implied from assignment of truth values to $\mathcal{X}_{a,v}^l$ through constraints (46). Satisfaction of constraints (47) is ensured by satisfaction of constraints (41) and by transitivity of

implication through the auxiliary $\mathcal{E}_u^l$. Connecting initial positions of agents with their goals by paths ensures satisfaction of constraints (42)-(45) enforcing that initial time layer and the final time layer correspond to initial and goal arrangements of agent respectively.

If on the other hand we have satisfying valuation of $F_{DIR}(\eta, \Sigma)$, non-overlapping vertex disjoint paths can be constructed from it. Paths $\pi_1, \pi_2, \dots, \pi_\mu$ will be constructed by following variables $\mathcal{X}_{a,v}^l$. Let $\pi_i = ([u_0, 0], [u_1, 1], [u_2, 2], \dots, [u_\eta, \eta])$ where $\mathcal{X}_{a_i,u_0}^0$, $\mathcal{X}_{a_i,u_1}^1$, ..., $\mathcal{X}_{a_i,u_\eta}^\eta$ are $TRUE$. Single path is correctly defined as in each time layer $l \in \{0,1,\dots,\eta\}$ and agent $a_i \in A$ there is exactly one $\mathcal{X}_{a_i,v}^l$ with $v \in V$ set to $TRUE$ which is ensured by constraints (38). Consecutive vertices in the path are connected by arcs which is ensured by constraints (40). If we consider all the paths together, then constraints (39) enforce that paths never intersects because two distinct agents cannot share a vertex. Finally, non-overlapping is ensured by constraints (41) since whenever non-trivial traversal between two consecutive time layers is made, no other agent can be involved in affected vertices. ■

Again, recall that non-overlapping vertex disjoint paths correspond to CPF solutions (Proposition 1) which together with just proven result gives the following theorem.

Theorem 4 ***(SOLUTION OF*** Σ ***AND*** $F_{DIR}(\eta, \Sigma)/F_{SIM}(\eta, \Sigma)$ ***SATISFACTION).*** *A solution of a CPF* $\Sigma = (G, A, \alpha_0, \alpha_+)$ *exists if and only if there exist* $\eta \in \mathbb{N}$ *for that formula* $F_{DIR}(\eta, \Sigma)$ *is satisfiable. The same result holds for the simplified formula* $F_{SIM}(\eta, \Sigma)$. ■

4.3. Summary of the Size Complexity of Propositional Encodings

Theoretical analysis of the size of encodings has been fine grained so far and it is not straightforward to see immediately how individual encodings compare with each other just by looking on expressions. Therefore, the extreme cases of all the expressions showing the number of variables and clauses have been evaluated and are shown here to provide a more complete picture.

The extreme cases concern the number of agents and neighborhood size in the graph G, which is either considered to be constant or asymptotically the same as the number of vertices.

Assumptions that the number of agents μ and the size of neighborhood in the graph asymptotically compares the number of vertices has been adopted in the

space analysis in order to show the number of variables and clauses as much as possible in terms of the size of the input graph $G = (V, E)$.

Thus, we have following 4 scenarios (2 cases for each of 2 parameters):

- **Scenario (i):** The number of agents μ and the size of the neighborhood in G is asymptotically the same as the number of vertices. (that is, $\mu \in \Theta(|V|)$ and $\deg_G(v) \in \Theta(|V|)$ for $\forall v \in V$).

  The assumption tells that the graph is highly occupied by agents and that the graph contains many edges. The consequence of the second assumption is also that the number of edges in the graph is asymptotically quadratic with respect to the number of vertices; that is, $|E| \in \Theta(|V|^2)$.

  Space complexities in terms of the number of variables and clauses based upon above assumptions for this scenario are shown in Table 1.

Table 1. *Comparison of the Size Complexities of CPF Encodings – **Scenario (i)**.* The number of agents μ in this scenario is asymptotically the same as the number of vertices of G (that is, $\mu \in \Theta(|V|)$) and the size of the vertex neighborhood in G is also asymptotically the same as the number of vertices (that is, $\deg_G(v) \in \Theta(|V|)$ for $\forall v \in V$). For reference fine-grained complexity expression are shown as well.

	#Variables fine-grained/scenario (i)	#Clauses fine-grained/scenario (i)
INVERSE $F_{INV}(\eta, \Sigma)$	$O(\eta \cdot (\lvert V\rvert \cdot \lceil\log_2(\mu)\rceil + \sum_{v\in V}\lceil\log_2(\deg_G(v))\rceil + \lvert E\rvert))$	$O(\eta \cdot (\lvert V\rvert \cdot \lceil\log_2(\mu)\rceil + \lvert E\rvert \cdot \lceil\log_2(\mu)\rceil + \sum_{v\in V}\deg_G(v) \cdot (\lceil\log_2(\deg_G(v))\rceil))$
	$O(\eta \cdot \lvert V\rvert^2)$	$O(\eta \cdot \lvert V\rvert^2 \cdot \lceil\log_2\lvert V\rvert\rceil)$
ALL-DIFFERENT $F_{DIFF}(\eta, \Sigma)$	$O(\eta \cdot \mu \cdot \lvert V\rvert)$	$O\left(\eta \cdot \lceil\log_2\lvert V\rvert\rceil \cdot \left(\binom{\mu}{2} + \mu \cdot \lvert V\rvert\right)\right)$
	$O(\eta \cdot \lvert V\rvert^2)$	$O(\eta \cdot \lvert V\rvert^2 \cdot \lceil\log_2\lvert V\rvert\rceil)$
MATCHING $F_{MATCH}(\eta, \Sigma)$	$O(\eta \cdot (\lvert E\rvert + \lvert V\rvert \cdot \lceil\log_2(\mu)\rceil))$	$O\left(\eta \cdot \left((\lvert V\rvert + \lvert E\rvert) \cdot \lceil\log_2(\mu)\rceil + \sum_{v\in V}\binom{\deg_G(v)}{2}\right)\right)$
	$O(\eta \cdot \lvert V\rvert^2)$	$O(\eta \cdot \lvert V\rvert^3)$
DIRECT $F_{DIR}(\eta, \Sigma)$	$O(\eta \cdot \mu \cdot \lvert V\rvert)$	$O(\eta \cdot (\mu \cdot \lvert V\rvert^2 + \mu^2 \cdot \lvert V\rvert + \mu^2 \cdot \lvert E\rvert))$
	$O(\eta \cdot \lvert V\rvert^2)$	$O(\eta \cdot \lvert V\rvert^4)$
SIMPLIFIED $F_{SIM}(\eta, \Sigma)$	$O(\eta \cdot \mu \cdot \lvert V\rvert)$	$O(\eta \cdot (\mu \cdot \lvert V\rvert^2 + \mu^2 \cdot \lvert V\rvert + \mu \cdot \lvert E\rvert))$
	$O(\eta \cdot \lvert V\rvert^2)$	$O(\eta \cdot \lvert V\rvert^3)$

- **Scenario (ii):** The number of agents μ is asymptotically the same as the number of vertices while the size of the neighborhood in G is asymptotically constant (that is, $\mu \in \Theta(|V|)$ and $\deg_G(v) \in \Theta(1)$ for $\forall v \in V$).

The second assumption tells the graph is sparse and can be intuitively compared to *planar graphs* [36] that are very common in practice. The assumption also tells that the number of edges is asymptotically the same as the number of vertices; that is, $|E| \in \Theta(|V|)$.

Space complexities for this scenario are shown in Table 2.

Table 2. *Comparison of the size complexities of CPF encodings –* ***Scenario (ii)****.* The number of agents μ in this scenario is asymptotically the same as the number of vertices of G (that is, $\mu \in \Theta(|V|)$) while the size of the vertex neighborhood in G is asymptotically constant (that is, $\deg_G(v) \in \Theta(1)$ for $\forall v \in V$).

	#Variables scenario (ii)	#Clauses scenario (ii)
INVERSE $F_{INV}(\eta, \Sigma)$	$\mathcal{O}(\eta \cdot \lvert V \rvert \cdot \lceil \log_2 \lvert V \rvert \rceil)$	$\mathcal{O}(\eta \cdot \lvert V \rvert \cdot \lceil \log_2 \lvert V \rvert \rceil)$
ALL-DIFFERENT $F_{DIFF}(\eta, \Sigma)$	$\mathcal{O}(\eta \cdot \lvert V \rvert^2)$	$\mathcal{O}(\eta \cdot \lvert V \rvert^2 \cdot \lceil \log_2 \lvert V \rvert \rceil)$
MATCHING $F_{MATCH}(\eta, \Sigma)$	$\mathcal{O}(\eta \cdot \lvert V \rvert \cdot \lceil \log_2 \lvert V \rvert \rceil)$	$\mathcal{O}(\eta \cdot \lvert V \rvert \cdot \lceil \log_2 \lvert V \rvert \rceil)$
DIRECT $F_{DIR}(\eta, \Sigma)$	$\mathcal{O}(\eta \cdot \lvert V \rvert^2)$	$\mathcal{O}(\eta \cdot \lvert V \rvert^3)$
SIMPLIFIED $F_{SIM}(\eta, \Sigma)$	$\mathcal{O}(\eta \cdot \lvert V \rvert^2)$	$\mathcal{O}(\eta \cdot \lvert V \rvert^3)$

- **Scenario (iii):** The number of agents μ is asymptotically constant while the size of the neighborhood in G is asymptotically the same as the number of vertices.
(that is, $\mu \in \Theta(1)$ and $\deg_G(v) \in \Theta(|V|)$ for $\forall v \in V$).

This scenario can be intuitively regarded as a planar graph densely occupied by agents. Space complexities for this scenario are shown in Table 3.

- **Scenario (iv):** The number of agents μ and the size of the neighborhood in G are both asymptotically constant.
(that is, $\mu \in \Theta(1)$ and $\deg_G(v) \in \Theta(1)$ for $\forall v \in V$).

Again, this scenario can be intuitively regarded as a planar graph with few agents inside. Space complexities for this scenario are shown in Table 4.

The measure used here for comparison of encodings is that the smaller number of variables or clauses the better. This is usually a realistic measure as the search space often correlates with the number of (decision) variables when solving the propositional formula satisfiability problem by standard search procedures. Similarly, the small number of clauses means that the overall size of the propositional formula is small and thus it is easier for the overall processing. Nevertheless, such small formula preference should be considered just as an intuitive measure since sometimes lot of variables may be derivable from values of other variables (thus they do not increase size of the search space) and sometimes more clauses may improve propagation.

Table 3. *Comparison of the size complexities of CPF encodings –* ***Scenario (iii)****.* The number of agents μ in this scenario is constant (that is, $\mu \in \Theta(1)$) while the size of the vertex neighborhood is asymptotically the same as the number of vertices of G (that is, $\deg_G(v) \in \Theta(|V|)$ for $\forall v \in V$).

	#Variables scenario (iii)	#Clauses scenario (iii)
INVERSE $F_{INV}(\eta, \Sigma)$	$\mathcal{O}(\eta \cdot \lvert V\rvert^2)$	$\mathcal{O}(\eta \cdot \lvert V\rvert^2 \cdot \lceil \log_2 \lvert V\rvert \rceil)$
ALL-DIFFERENT $F_{DIFF}(\eta, \Sigma)$	$\mathcal{O}(\eta \cdot \lvert V\rvert)$	$\mathcal{O}(\eta \cdot \lvert V\rvert \cdot \lceil \log_2 \lvert V\rvert \rceil)$
MATCHING $F_{MATCH}(\eta, \Sigma)$	$\mathcal{O}(\eta \cdot \lvert V\rvert^2)$	$\mathcal{O}(\eta \cdot \lvert V\rvert^3)$
DIRECT $F_{DIR}(\eta, \Sigma)$	$\mathcal{O}(\eta \cdot \lvert V\rvert)$	$\mathcal{O}(\eta \cdot \lvert V\rvert^2)$
SIMPLIFIED $F_{SIM}(\eta, \Sigma)$	$\mathcal{O}(\eta \cdot \lvert V\rvert)$	$\mathcal{O}(\eta \cdot \lvert V\rvert^2)$

Several conclusions can be made upon asymptotic numbers of variables and clauses in individual encodings presented in Table 1 - Table 4. In cases with many agents and dense graphs (corresponding to scenario (i)), INVERSE and ALL-DIFFERENT encodings excel in a small number of clauses.

When we have many agents and relatively sparse graphs (scenario (ii)), which is the most common case in practice, then INVERSE and MATCHING encodings excel in both, in the number of variables as well as in the number of clauses.

The remaining two scenarios (scenario (iii) and (iv)) can be regarded as non-cooperative scenarios since the number of agents is constant and hence the interaction among them is limited. The ALL-DIFFERENT encoding is the most space saving in a case with dense graphs (scenario (iii)) while INVERSE and MATCHING encodings are the most space saving on sparse graphs (scenario (iv)).

Observe that DIRECT and SIMPLIFIED encodings do not excel in any of the suggested scenarios. This is mostly caused by the fact that no binary encoding of finite domain state variables, which significantly reduces the size of representation of the state variable using propositional variables, is used in these two encodings.

Table 4. *Comparison of the size complexities of CPF encodings –* ***Scenario (iv)***. Both the number of agents μ as well as the size of the vertex neighborhood are asymptotically constant in this scenario (that is, $\mu \in \Theta(1)$ and $\deg_G(v) \in \Theta(1)$ for $\forall v \in V$).

	#Variables scenario (iv)	#Clauses scenario (iv)
INVERSE $F_{INV}(\eta, \Sigma)$	$\mathcal{O}(\eta \cdot \lvert V\rvert)$	$\mathcal{O}(\eta \cdot \lvert V\rvert)$
ALL-DIFFERENT $F_{DIFF}(\eta, \Sigma)$	$\mathcal{O}(\eta \cdot \lvert V\rvert)$	$\mathcal{O}(\eta \cdot \lvert V\rvert \cdot \lceil \log_2 \lvert V\rvert \rceil)$
MATCHING $F_{MATCH}(\eta, \Sigma)$	$\mathcal{O}(\eta \cdot \lvert V\rvert)$	$\mathcal{O}(\eta \cdot \lvert V\rvert)$
DIRECT $F_{DIR}(\eta, \Sigma)$	$\mathcal{O}(\eta \cdot \lvert V\rvert)$	$\mathcal{O}(\eta \cdot \lvert V\rvert^2)$
SIMPLIFIED $F_{SIM}(\eta, \Sigma)$	$\mathcal{O}(\eta \cdot \lvert V\rvert)$	$\mathcal{O}(\eta \cdot \lvert V\rvert^2)$

4.3.1. Knowledge Compilation – Distance Heuristics

Encodings based on time expansion graph can be be further enhanced by a so called *distance heuristic*. Intuitively said, a path indicating the trajectory of a given agent cannot go through vertices that there too far from the initial or the goal vertex under given time constraints. In other words, vertices at a given time layer where the distance to the initial position of the agent is larger than the time elapsed for the time layer (which equals to the position of the time layer in the time expansion graph) or where the distance to the goal vertex is larger than the time that remains for the given time layer (which equals to the position of the time layer in the time expansion graph counted from the end) can never be visited by

the agent. The just described time consideration can be easily formalized in the time expansion graphs through existence of directed paths.

The knowledge of these impassable vertices can rule out the occurrence of the agent in them from further consideration during the search for a solution and consequently reduce the search space.

Assume a time expansion graph $\text{Exp}_\text{T}(G,\eta)$ for CPF $\Sigma = (G, A, \alpha_0, \alpha_+)$; let $\text{dist}_\text{D}^{\rightarrow}(u,v)$ denote the length of the shortest directed path connecting u to v in a given digraph $D = (X,F)$; $\text{dist}_\text{D}^{\rightarrow}(u,v) = \infty$ if there is no path connecting u to v in D.

Proposition 10 ***(DISTANCE HEURISTIC).*** *Any solution* $\vec{s} = [\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_\eta]$ *to* Σ *satisfies that* $\text{dist}_{\text{Exp}_\text{T}(G,\eta)}^{\rightarrow}([\alpha_0(a_i),0],[\alpha_l(a_i),l]) < \infty$ *and* $\text{dist}_{\text{Exp}_\text{T}(G,\eta)}^{\rightarrow}([\alpha_l(a_i),l], [\alpha_\eta(a_i),\eta]) < \infty$ *for every* $i \in \{1,2,\ldots,\mu\}$ *and* $l \in \{0,1,\ldots,\eta\}$. ■

Proof. The proposition is in fact a direct consequence of Proposition 1. If $\text{dist}_{\text{Exp}_\text{T}(G,\eta)}^{\rightarrow}([\alpha_0(a_i),0],[v,l]) = \infty$ or $\text{dist}_{\text{Exp}_\text{T}(G,\eta)}^{\rightarrow}([v,l],[\alpha_\eta(a_i),\eta]) = \infty$ for some $v \in V$ then there is no directed path connecting $[\alpha_0(a_i),0]$ and $[\alpha_\eta(a_i),\eta]$ going through $[v,l]$. A fortiori, there is no path connecting $[\alpha_0(a_i),0]$ and $[\alpha_\eta(a_i),\eta]$ visiting $[v,l]$ that does not overlap and does not intersect other paths. Hence, $\alpha_l(a_i) \neq v$. ■

The above proposition can be used to design a heuristic. All the vertices $[v_j,l]$ with $v_j \in V$ and $l \in \{0,1,\ldots,\eta\}$ in $\text{Exp}_\text{T}(G,\eta)$ for that $\text{dist}_{\text{Exp}_\text{T}(G,\eta)}^{\rightarrow}([\alpha_0(a_i),0],[v_j,l]) = \infty$ or $\text{dist}_{\text{Exp}_\text{T}(G,\eta)}^{\rightarrow}([v_j,l],[\alpha_\eta(a_i),\eta]) = \infty$ can be excluded from trajectories corresponding to agent a_i (in original graph it translates to the requirement that agent a_i cannot enter v_j at time step l). In all the encodings, this can be done easily as follows:

$$\mathcal{A}_{v_j}^{l} \neq i \quad \text{in the INVERSE and MATCHING encoding} \tag{30}$$

$$\mathcal{L}_{a_i}^{l} \neq j \quad \text{in the ALL-DIFFERENT encoding} \tag{31}$$

$$\neg\mathcal{X}_{a_i,v_j}^{l} \quad \text{in the DIRECT/SIMPLIFIED encoding} \tag{32}$$

The inequality between a bit vector and a constant is encoded as a single clause that forbids the bit vector to take that constant. That is, at least one bit must

disagree with binary representation of the constant. For example, the inequality $\mathcal{A}_v^l \neq c$ is encoded as follows:

$$\text{con}_{\neq}(\mathcal{A}_v^l, c) = \bigvee_{\mathbb{i}=0}^{\lceil \log_2 \mu \rceil - 1} \neg\text{lit}(\mathcal{A}_v^l, c, \mathbb{i}) \tag{33}$$

It holds that added inequalities are logical consequences of the encoded propositional formulae (that is, for example $F_{INV}(\eta, \Sigma) \Rightarrow \mathcal{A}_{v_j}^l \neq i$ is a valid formula). Thus in theory, the SAT solver should be able to infer that some vertices are not reachable at certain time steps. However, it may be costly to derive such a fact for the SAT solver while the same knowledge can be obtained easily in advance and compiled directly into the formula almost without any increase of its complexity.

5. Experimental Evaluation

Experimental evaluation has been focused on measuring the actual size of suggested encodings and on measuring runtime when encodings are used for makespan optimal CPF solving.

The solving procedure presented as Algorithm 1 was used as a core framework for our makespan optimal CPF solving technique (that is, the sequential increasing strategy for querying the SAT solver was used) while suggested individual propositional encodings can be regarded as its exchangeable modules. The SAT solver itself was connected to the solving technique as another external module. All the implemented encodings used build-in distance heuristic discussed in section 4.3.1.

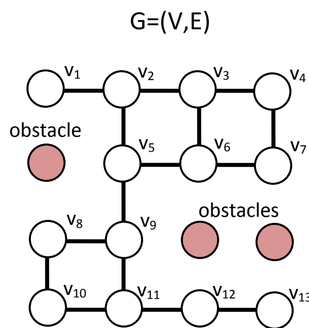


Figure 6. *Four-connected grid of size 4×4 with 3 obstacles.* Positions of obstacles within the grid are depicted though they are actually not present in the graph.

The SAT-based CPF solving procedure was implemented in C++ as well as procedures for generating propositional formulae from given CPF and makespan bound (solving procedure and formulae generation were compiled together as a single executable program).

We used `glucose 3.0` SAT solver [2] in our tests, which is justified by the fact that this SAT solver ranked among the winners in recent SAT Competitions [3] in the category of hard combinatorial problems to which we consider CPF belongs as well. The SAT solver was a sep-

arate module and was called externally by the CPF solving procedure (the solving procedure always generated a formula into a file, which was then submitted as input to the SAT solver; the answer of the SAT solver was generated into another file, from which the procedure read it and further processed).

5.1. Benchmark Setup

We followed benchmark setup suggested by Silver in [20]. Four-connected grids of various sizes were used to model environments in testing instances. The size of grids ranged from 6×6 to 12×12 with 20% of randomly selected vertices occupied by obstacles (obstacle was represented by a missing vertex in grid – see Figure 6). Initial and goal arrangements of agents was random – the random arrangement of agents has been obtained by placing agents one by one while the position has been uniformly randomly picked from the remaining unoccupied vertices. Only solvable instances were taken into runtime tests.

To allow full reproducibility of presented results all the source codes and experimental data were posted on-line on: http://ktiml.mff.cuni.cz/~surynek/research/j-encoding-2015.

5.2. Encoding Size Evaluation

The size of propositional formulae was tested for discussed 4-connected grids with the increasing number of agents inside. The number of agents ranged from 1 up to the half all the vertices in the graph.

For each number of agents, 10 random CPF instances were generated and their characteristics were measured. Formulae corresponding to all the suggested encodings were generated for each number of agents. The number of layers in time expansion graphs was fixed and set relatively to the size of the instance – it was 12 for 6×6 grid; 16 for 8×8 grid; and 24 for 12×12 grid.

The average number of propositional variables, average number of clauses, ratio between the number of clauses and variables, and average clause length were calculated for each encoding and number of agents out of 10 randomly generated instances. Partial results are shown in Table 5, Table 6, and Table 7 – preferred values of individual characteristics are listed in bold.

The number of variables and clauses directly correspond to the size of formulae. Preference is given to formulae that are smaller as they are expected to be easier to solve as well as easier for processing.

The ratio of the number of clauses and the number of variables is an important measure of the difficulty of propositional formula. Formulae that are *under-constrained* or *over-constrained* are easier to solve [7] (easily satisfiable or easily

unsatisfiable respectively) and hence such situation is preferred in formulae encoding CPF.

Table 5. *Size comparison of propositional encodings of CPF over 6×6 grid.* CPF instances are generated over the 4-connected grid of size 6×6 with 20% of vertices occupied by obstacles. The number of time layers in corresponding time expansion graph η is 12. The number of *variables* and *clauses*, the *ratio* of the number of clauses and the number of variables, and the *average clause length* are listed for different sizes of the set of agents A. Small size of the formula and short clauses (they support unit propagation) are preferred – best values for each measure according to this preference are shown in bold. DIRECT and SIMPLIFIED encodings are best in number of measures on 6×6 grid.

Grid 6×6 |Agents|		INVERSE		ALL-DIFFERENT		MATCHING		DIRECT		SIMPLIFIED	
1	#Variables / *Ratio*	3 384.3	*3.692*	701.4	*4.506*	1 841.1	*5.595*	**342.0**	*17.685*	684.0	*2.192*
	#Clauses / *Length*	12 494.3	*2.622*	3 160.7	*2.979*	10 300.6	*2.436*	6 048.2	***2.261***	**1 499.6**	*2.587*
2		3 738.3	*4.551*	1 723.5	*4.173*	2 195.1	*6.149*	**684.0**	*20.725*	1 026.0	*3.354*
		17 012.0	*2.599*	7 191.7	*2.980*	13 497.1	*2.512*	14 176.0	*2.353*	**3 441.4**	*2.562*
4		4 092.3	*5.403*	4 127.5	*3.729*	2 549.1	*6.777*	**1 368.0**	*25.558*	1 710.0	*4.653*
		22 110.2	*2.642*	15 392.6	*3.026*	17 274.1	*2.632*	34 962.7	*2.427*	**7 956.1**	***2.423***
8		4 446.3	*6.348*	12 066.7	*3.250*	2 903.1	*7.601*	**2 736.0**	*36.324*	3 078.0	*6.964*
		28 225.0	*2.794*	39 216.1	*3.060*	22 067.7	*2.867*	99 381.3	*2.543*	**21 436.3**	***2.319***
16		4 800.3	*7.609*	38 791.0	*2.830*	**3 257.1**	*8.919*	5 472.0	*56.375*	5 814.0	*11.062*
		36 527.1	*3.133*	109 781.6	*3.104*	**29 048.6**	*3.313*	308 484.7	*2.633*	64 314.8	***2.201***

A very important characteristic is the average length of clause while short clauses are preferred since they support *unit propagation* [7], which allows deriving values for other variable without search.

Table 6. *Size comparison of propositional encodings of CPF over 8×8 grid.* The number of time layers in the corresponding time expansion graph is 16. DIRECT and SIMPLIFIED encodings have fewer variables and clauses for small number of agents while MATCHING encoding is better in these measures for many agents in the instance.

Grid 8×8 |Agents|		INVERSE		ALL-DIFFERENT		MATCHING		DIRECT		SIMPLIFIED	
1	#Variables / *Ratio*	4 520.3	*3.748*	1 489.3	*5.325*	4 520.3	5.710	**814.4**	*28.539*	1 628.8	*2.078*
	#Clauses / *Length*	25 881.1	*2.616*	7 930.4	*3.057*	25 881.1	2.441	23 241.9	***2.149***	**3 384.6**	*2.550*
4		10 019.5	*5.532*	7 834.5	*4.440*	6 181.1	6.984	**3 257.6**	*35.589*	4 072.0	*4.420*
		55 437.0	*2.641*	34 781.9	*3.103*	43 171.0	2.640	115 934.3	***2.272***	**17 997.8**	*2.374*
8		10 849.9	*6.519*	21 875.4	*3.831*	7 011.5	7.851	**6 515.2**	*45.635*	7 329.6	*5.736*
		70 725.9	*2.792*	83 794.2	*3.113*	55 050.3	2.874	297 319.9	***2.390***	**49 381.3**	*2.694*
16		11 680.3	*7.820*	67 088.3	*3.231*	**7 841.9**	9.215	13 030.4	*64.506*	13 844.8	*10.853*
		91 344.5	*3.127*	216 745.4	*3.147*	**72 259.3**	3.315	840 540.6	*2.505*	150 259.2	***2.180***
32		12 510.7	*9.765*	230 753.0	*2.802*	**8 672.3**	11.494	26 060.8	*105.084*	26 875.2	*19.002*
		122 170.3	*3.733*	646 616.2	*3.168*	**99 675.5**	4.045	2 738 584.7	*2.621*	510 672.1	***2.111***

Results indicate that DIRECT and SIMPLIFIED encodings have best size characteristics with respect to the small size preference in cases with small number of agents in the instance. This result can be observed for all the sizes of the grid modeling the environment. As the neighborhood connectivity in 4-connected grids can be regarded as constant; that is, $\deg_G(v) \in \Theta(1)$ for $\forall v \in V$, the cases, where DIRECT and SIMPLIFIED encoding have best size characteristics, rough-

ly correspond to scenario (iv). However, theoretical asymptotic formula size estimations suggest different results - DIRECT and SIMPLIFIED encodings should be same as other encodings in terms of the number of variables and worse than other encodings in terms of the number of variables. Hence, experimental evaluation has shown a surprising result in this aspect.

If the number of agents is higher, the MATCHING encoding dominates in the size characteristics for all the size of the grid. It has the fewest number of propositional variables as well as the fewest number of clauses. If we consider that this case roughly correspond to scenario (ii), these observations correspond to theoretical asymptotic estimations, which indicate that MATCHING encoding together with INVERSE encoding should be smallest (note, that the INVERSE encoding is the second smallest according to experimental results).

Table 7. *Size comparison of propositional encodings of CPF over 12×12 grid.* The number of time layers in the time expansion graph is 24 here. The MATCHING encoding is clearly the smallest encoding for larger number of agents.

Grid 12×12 / |Agents|			INVERSE		ALL-DIFFERENT		MATCHING		DIRECT		SIMPLIFIED	
1	#Variables	*Ratio*	29 798.7	*3.903*	4 973.9	*6.218*	15 961.3	*5.927*	**2 767.2**	*60.721*	5 534.4	*2.094*
	#Clauses	*Length*	116 302.8	*2.635*	30 928.8	*3.031*	94 603.2	*2.443*	168 027.8	***2.073***	**11 587.0**	*2.578*
8			38 172.3	*6.752*	55 602.1	*4.887*	24 334.9	*8.130*	**22 137.6**	*77.789*	24 904.8	*6.707*
			257 739.9	*2.793*	271 730.3	*3.088*	197 835.9	*2.871*	1 722 059.3	***2.230***	167 026.1	*2.289*
16			40 963.5	*8.062*	153 047.5	*4.290*	**27 126.1**	*9.510*	44 275.2	*97.349*	47 042.4	*11.540*
			330 249.1	*3.115*	656 615.4	*2.999*	257 974.6	*3.300*	4 310 137.7	*2.343*	542 862.4	***2.059***
32			43 754.7	*10.049*	475 135.0	*3.428*	**29 917.3**	*11.843*	88 550.4	*136.888*	91 317.6	*18.953*
			439 680.0	*3.701*	1 628 634.8	*3.148*	**354 306.4**	*4.021*	12 121 528.6	*2.475*	1 730 745.7	***2.112***
64			46 545.9	*13.340*	1 626 205.9	*2.898*	**32 708.5**	*15.985*	177 100.8	*216.610*	179 868.0	*35.0127*
			620 942.7	*4.632*	4 713 520.1	*3.183*	**522 834.3**	*5.065*	38 361 723.7	*2.594*	6 297 660.9	***2.062***

DIRECT and SIMPLIFIED encodings excel in terms of the ratio of the number of clauses to the number of variables. Both encodings tend to be overconstrained, which intuitively suggest easier proving of unsatisfiability. The average length of clauses is shortest for the SIMPLIFIED encoding. As the number of agents increases the average clause length converges towards 2 for all the sizes of the grid (that is, most of clauses are binary in the SIMPLIFIED encoding).

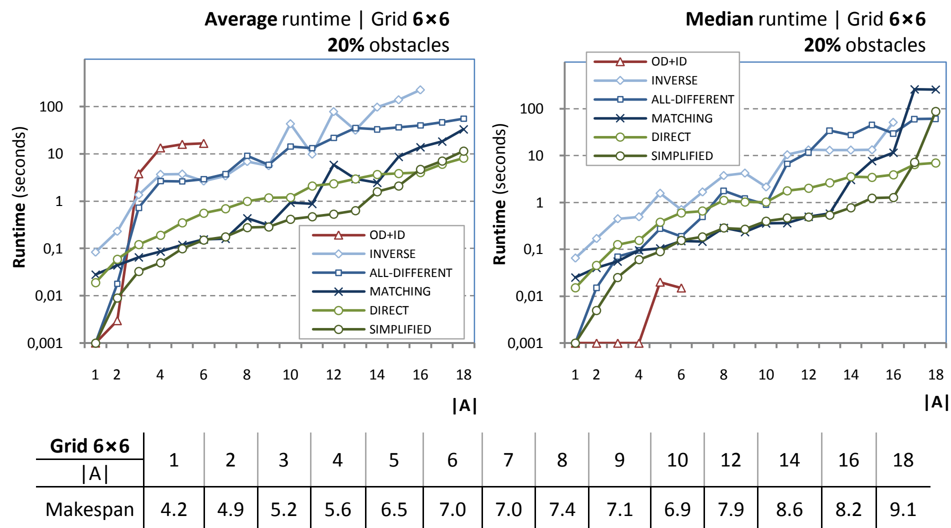
Above observations of static characteristics of encodings indicate that MATCHING encoding and especially SIMPLIFIED encoding should perform well in CPF solving (or at least better than other encodings).

5.3. Runtime Evaluation

We re-implemented A*-based OD+ID CPF solving procedure [23] in C++ with the objective function for minimizing the makespan and compared it with our

SAT based solving method in order to provide broader picture regarding the runtime evaluation.

Again, CPFs over 4-connected grids of sizes 6×6, 8×8, and 12×12 with 20% of vertices occupied by randomly placed obstacles were used. Initial and goal arrangements of agents were generated randomly. Runtime evaluation was done for the increasing number of agents in instances while for each number of agents 10 random instances were generated and solved. All the instances used for evaluation were solvable.



Grid 6×6 |A|	1	2	3	4	5	6	7	8	9	10	12	14	16	18
Makespan	4.2	4.9	5.2	5.6	6.5	7.0	7.0	7.4	7.1	6.9	7.9	8.6	8.2	9.1

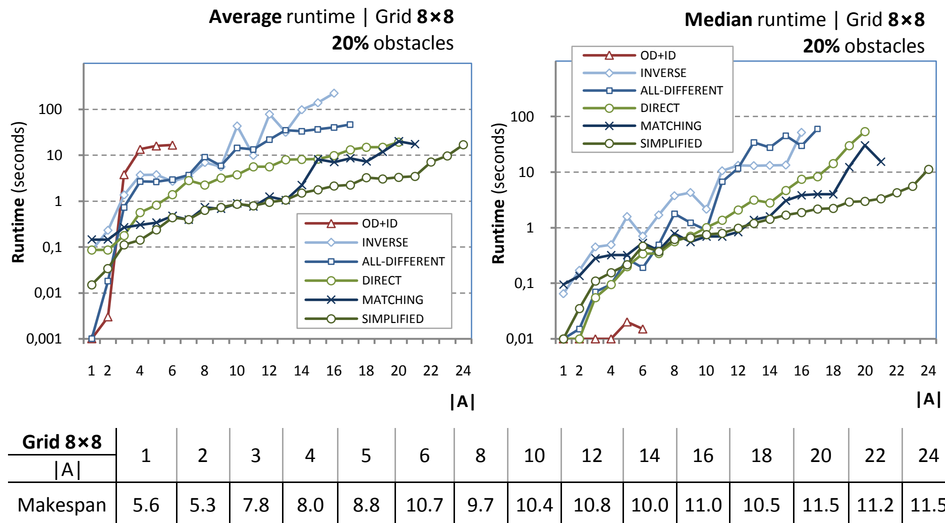
Figure 7. ***Runtime evaluation*** *over 6×6 4-connected grid with 20% of vertices occupied by obstacles.* A*-based method OD+ID and SAT-based method with INVERSE, ALL-DIFFERENT, MATCHING, DIRECT, and SIMPLIFIED encodings are compared on random CPF instances on the grid. Average and median runtimes out of runtimes on 10 random instances are shown; average optimal makespans are also shown. OD+ID method does not scale for higher number of agents while SAT-based solving performs better with many agents. Particularly SIMPLIFIED encoding performs as best. Up to two orders of magnitude are between the best and worst encoding in runtime.

The timeout for single CPF instance solving was set to 256 seconds (approximately 4 minutes). The number of agents was increased until all the 10 random instances were solvable within the given timeout – that is, each solving method (encoding) is characterized by the maximum number of agents for which it is able to solve all the 10 random instances within the given timeout.

The average and median runtimes were calculated out of these 10 instances for all the tested methods. In the case of SAT based CPF solving methods, the runtime is a sum of the runtime of the core CPF solving procedure (corresponding to

Algorithm 1) plus runtimes of all the runs of the SAT solver invoked by the core procedure.

Runtime results together with average optimal makespan are shown in Figure 7, Figure 8, and Figure 9[1] (note that, all the methods generate solutions of the same optimal makespan).



Grid 8×8 |A|	1	2	3	4	5	6	8	10	12	14	16	18	20	22	24
Makespan	5.6	5.3	7.8	8.0	8.8	10.7	9.7	10.4	10.8	10.0	11.0	10.5	11.5	11.2	11.5

Figure 8. ***Runtime evaluation*** *over 8×8 4-connected grid with 20% of vertices occupied by obstacles.* Again, the SAT-based solving with SIMPLIFIED encoding performs as best for higher number of agents. The MATCHING encoding also starts with promising performance but it quickly degrades for more than approximately 14 agents.

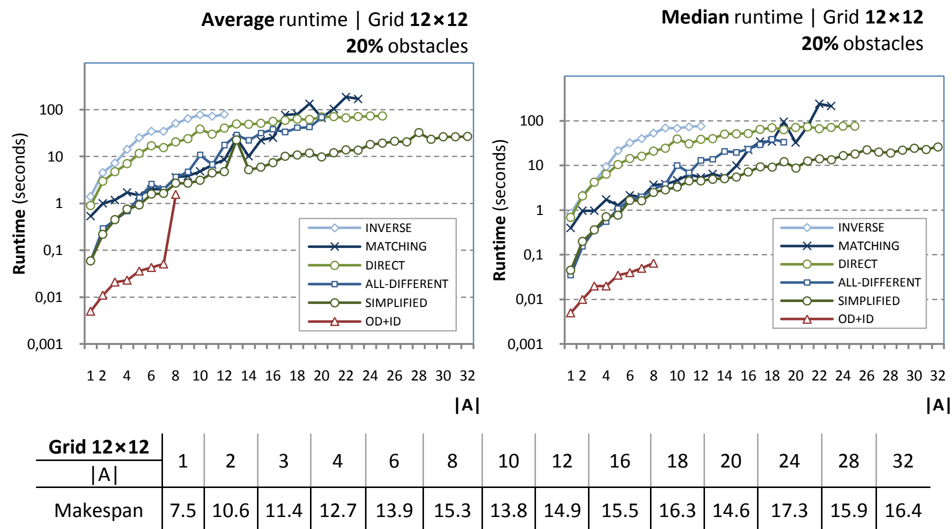
It can be observed that OD+ID, although it is the fastest for small number of agents, does not scale up as the runtime quickly blows up for more agents. The SAT-based solving method with all the encodings performs better and scales up for higher number of agents. Particularly, the SIMPLIFIED encoding performs as best in all the sizes of the grid followed by MATCHING, DIRECT, ALL-DIFFERENT, and INVERSE encodings respectively.

Note that the good performance of the SIMPLIFIED encoding has been predicted by the static analysis of encodings (particularly, it has been assumed to support unit propagation well). Another well competing MATCHING encoding

[1] All the runtime measurements were done on a machine with the 4-core CPU Xeon 2.0GHz and 12GB RAM under Linux kernel 3.5.0-48. Although we used multiple cores to run experiments in parallel, the individual instances were solved in a single thread (that is, the core solving procedure and all its call to the SAT solver were run in single thread).

had been predicted to have a good performance as well due to its small size in testing instances.

An interesting behavior can be observed with MATCHING encoding that start with almost the same promising performance as the SIMPLIFIED encoding for small number of agents, but it quickly degrades and it is eventually outperformed by the DIRECT encoding on 6×6 and 12×12 grids for higher number of agents (in case of the 8×8 grid, the degradation of the MATCHING encoding can be observed as well but it is less significant – the DIRECT encoding reached the timeout before it could overtake the MATCHING encoding).



Grid 12×12 |A|	1	2	3	4	6	8	10	12	16	18	20	24	28	32
Makespan	7.5	10.6	11.4	12.7	13.9	15.3	13.8	14.9	15.5	16.3	14.6	17.3	15.9	16.4

Figure 9. ***Runtime evaluation*** *over 12×12 4-connected grid with 20% of vertices occupied by obstacles.* The best performance is exhibited by the SIMPLIFIED encoding again. The MATCHING encoding is able to solve second highest number of agents in the given timeout.

Instances with occupancy by agents up to 62% are solvable within the given timeout in the 6×6 grid by using the SIMPLIFIED encoding. This figure is 46% for the 8×8 grid and 28% for the 12×12 grid for the SIMPLIFIED encoding. OD+ID method can solve instances with occupancy up to 24%, 13%, and 7% in 6×6, 8×8, 12×12 grids respectively. Thus, approximately 3 times more agents are solvable with SAT based method than with OD+ID in given testing instances.

The general conclusion from the above experimental evaluation is also that the binary encoding used for encoding finite domain state variables in the INVERSE, ALL-DIFFERENT, and MATCHING encoding contributes to the small size but it is questionable if it contributes the overall performance as these encodings clearly

performed worse than the DIRECT and SIMPLIFIED encodings that did not rely on the binary encoding.

On the other hand, the simple design of the DIRECT and SIMPLIFIED encodings is not at the expense of the performance of their solving. The simple design of variables allowed modeling constraints using short clauses that significantly support intensive unit propagation, which is most likely the key factor for the good performance of both encodings – especially in the case of the SIMPLIFIED encoding.

5.4. Solution Quality Evaluation

Although all the solutions generated by the suggested SAT based solving techniques are makespan optimal, that is, the best with respect to our objective function, they may differ in other aspects. Particularly important is the *total number of moves* performed by agents (also called a *sum of costs*) which can be regarded as the total energy consumed by agents to perform their movements. The total number of moves is also considered as an objective function in several approaches to CPF solving such as [21, 22]. Hence, it is interesting what do solutions generated by makespan optimal SAT solving look like with respect to the total number of moves despite the fact that this aspect has been completely disregarded in the design of propositional encodings of CPF.

The way in which a given problem is encoded into propositional formula greatly affects heuristics the SAT solver uses for selecting variables and their values. Values selected to satisfy the formula are then reflected in the CPF solution reconstructed from its satisfying valuation. Although not a rule, SAT solvers in their default settings usually prefer assigning $FALSE$ value if it is not more advantageous than to assign value $TRUE$.

Observe that only values assigned to visible propositional variables are directly reflected in the resulting CPF solution. Visible propositional variables in the suggested encodings are either part of a directly encoded state (DIRECT and SIMPLIFIED encodings) or part of a binary encoded bit vector (INVERSE, ALL-DIFFERENT, and MATCHING encodings).

Propositional variables within directly encoded state directly correspond to occupancy of a vertex or an edge by a fixed agent. Assignment of value $FALSE$ to a propositional variable of the directly encoded state corresponds to no occupancy by the given fixed agent. Complete no-occupancy appears if and only if all the propositional variables directly encoding the state are set to $FALSE$ for all the agents.

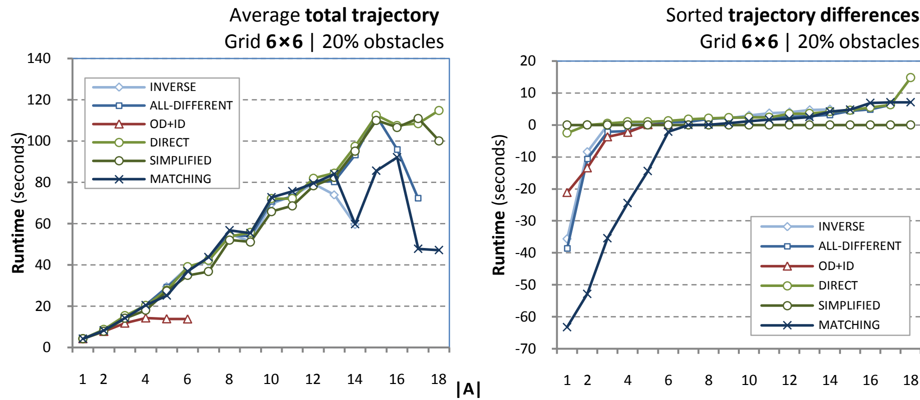


Figure 10. *Solution **quality comparison** over 6×6 4-connected grid.* The total number of moves in optimal solutions obtained by each tested method is compared for the growing number of agents in the grid (left part). Sorted differences in total number of moves from the number of moves generated with the SIMPLIFIED encoding are also shown (right part). The SIMPLIFIED encoding yields a solution with the fewer number of moves than other methods in about one third of all the generated solutions.

The interpretation of bit vector propositional variables is that occupancy of a corresponding vertex or an edge appears if any of the propositional variables within the bit vector is set to $TRUE$. No occupancy corresponds to the assignment of integer zero to the bit vector, which means to assign $FALSE$ to all the propositional variables, which the bit vector consists of.

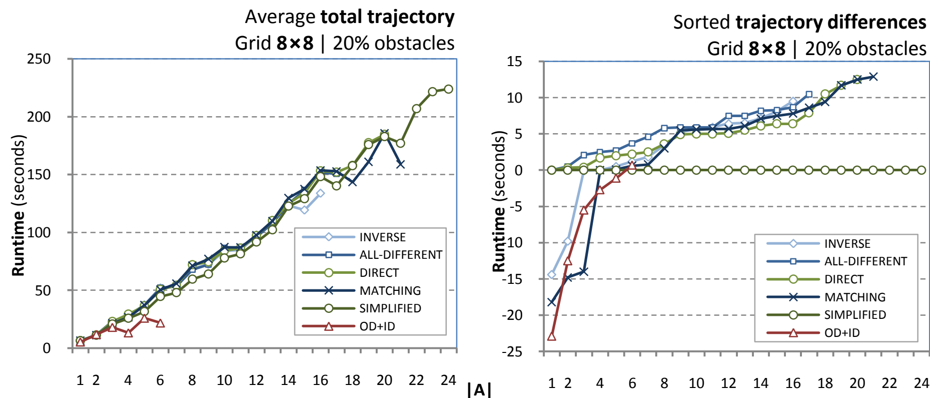


Figure 11. *Solution **quality comparison** over 8×8 4-connected grid.* The SIMPLIFIED encoding yields solutions with fewest moves in approximately 75% of cases of solution generation. Note that A*-based OD+ID generates solutions with even fewer moves but it does not scale up enough to show its qualities for higher density of agents.

If we assume that the SAT solver tries to find a solution conservatively; that is, it prefers to assign $FALSE$ values, then it seems that using encodings with visible variables that directly encode states results in smaller vertex and edge occupancy, which correspond to CPF solutions consisting of fewer total number of moves.

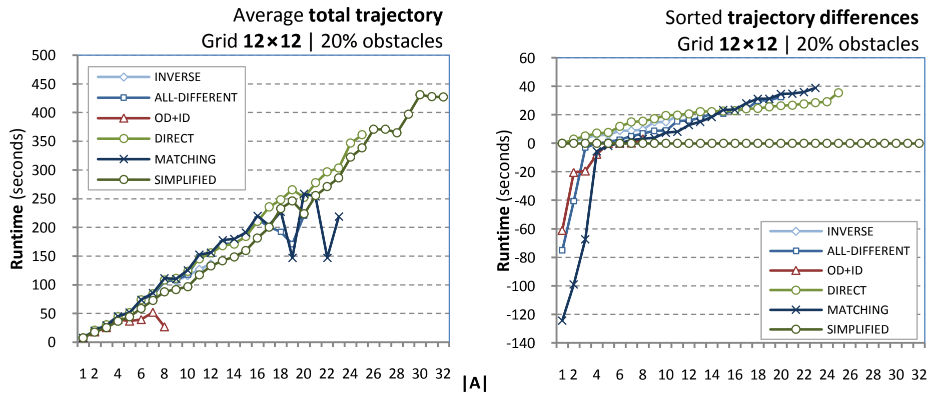


Figure 12. *Solution* ***quality comparison*** *over 12×12 4-connected grid.* The SIMPLIFIED encoding yields almost always a solution with fewer moves than other methods in this larger scenario. Again, solutions with fewest moves are generated by OD+ID but the comparison could be done for few agents only due to insufficient scalability of OD+ID.

The reasoning behind this hypothesis assumes to have a set of agents A and a location (vertex/edge) that is to be occupied by at most one agent from A. The occupancy of the location is modeled by directly encoded state in one scenario and as a binary encoded bit vector in the second scenario. The first scenario yields $|A|$ propositional variables with $|A| + 1$ allowed assignments - one of these assignments corresponding to no occupancy of the location assigns $FALSE$ to all the propositional variables; other allowed assignments have just a single propositional variable set to $TRUE$. The second scenario yields $\log_2\lceil|A| + 1\rceil$ propositional variables - all the possible combinations of Boolean values are allowed as assignments while all the propositional variables set to $FALSE$ correspond to no occupancy of the location. If the preference of assigning $FALSE$ actually results in setting strictly fewer variables to $TRUE$ then no occupancy immediately appears in the first scenario while there is little chance that no occupancy appears in the second scenario (setting strictly fewer variables to $TRUE$ may lead to another assignment of the bit vector with some propositional variables set to $TRUE$ - that is, representing some occupancy).

Results of measurement of the total number of moves generated by SAT based CPF solving with suggested encodings are presented in Figure 10, Figure 11, and Figure 12. The same set of testing instances over 4-connected grids as in the run-

time measurement has been used. The total number of moves generated by OD+ID is also included in the measurement.

The fewest number of moves in most testing instances is yielded by the SIMPLIFIED encoding. Thus, we also present sorted differences in the total number of moves between those yielded by the SIMPLIFIED encoding and other methods. It can be also observed that OD+ID generates solutions with the smallest number of moves in instances containing few agents. Unfortunately, as OD+ID does not scale up well enough it cannot show qualities of its solutions for larger number of agents.

There is almost no significant difference in the total number of moves generated by other methods except a marginal tendency of the DIRECT encoding to yield better solutions than methods using binary encoded bit vectors especially observable over 8×8 grid.

Altogether, we can conclude that the hypothesis that encodings using directly encoded states is more advantageous than encodings with binary encoded bit-vectors with respect to the total number of moves.

6. Conclusions

Several propositional encodings of cooperative path-finding problem (CPF) have been introduced - INVERSE, ALL-DIFFERENT, MATCHING, DIRECT, and SIMPLIFIED encodings. The presented encodings are based on the notion of the time expanded graph that expands the graph modeling the environment over time so that arrangements of agents at all the time steps up to a certain final time step can be represented. Time expanded graphs provided an essential step towards building propositional formulae, in which a query whether there is a solution of a given CPF with the specified number of time steps is encoded. Obtaining makespan optimal solution is then carried out by submitting multiple encoded queries to a SAT solver. The reduction of CPF to SAT allows accessing all the advanced search, pruning, and learning techniques of the SAT solver that can be in this way employed in CPF solving.

The suggested encodings either use binary encoded bit vectors (INVERSE, ALL-DIFFERENT, and MATCHING encoding) or directly encoded states (DIRECT and SIMPLIFIED encodings) to model arrangements of agents at individual time steps. Using binary encoded bit vectors results in smaller formulae in terms of the number of variables and clauses. The advantage of encodings with directly encoded states is on the other hand a better support for Boolean constraint propagation (unit propagation) which enabled by the presence of many short clauses.

Performed experimental evaluation indicates that CPF solving via SAT is generally the best option in highly constrained situations (environments densely occupied by agents). SAT based CPF solving scales up for larger number of agents much better than the alternative A* based search technique.

If we compare solely SAT encodings, than the SIMPLIFIED encoding turned out to perform as best. Instances with the highest occupancy by agents were solved only by the SIMPLIFIED encoding in the given timeout. Moreover, the comparison of the quality of solutions generated by the SAT based solving in terms of the total number of generated moves also indicates that the SIMPLIFIED encoding generates fewest moves.

References

1. **Ahuja**, R. K., **Magnanti**, T. L., **Orlin**, J. B. *Network flows: theory, algorithms, and applications*. Prentice Hall, 1993.
2. **Audemard**, G., **Simon**, L. *The Glucose SAT Solver.* http://labri.fr/perso/lsimon/glucose/, 2013, [accessed in September 2014].
3. **Balint**, A., **Belov**, A., **Heule**, M., and **Järvisalo**, M. *SAT 2013 competition.* http://www.satcompetition.org/, 2013, [accessed in April 2015].
4. **Balyo**, T., 2013. *Relaxing the Relaxed Exist-Step Parallel Planning Semantics.* Proceedings of the 26th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2013), pp 865-871, IEEE Press.
5. **Barer**, M., **Sharon**, G., **Stern**, R., **Felner**, A. *Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem.* ECAI 2014 - 21st European Conference on Artificial Intelligence (ECAI 2014), pp. 961-962, IOS Press, 2014.
6. **Biere**, A., **Brummayer**, R. *Consistency Checking of All Different Constraints over Bit-Vectors within a SAT Solver.* Proceedings of Formal Methods in Computer-Aided Design (FMCAD 2008), pp. 1-4, IEEE Press, 2008.
7. **Biere**, A., **Heule**, M., **van Maaren**, H., **Walsh**, T. *Handbook of Satisfiability.* IOS Press, 2009.
8. **Čáp**, M., **Novák**, P., **Vokřínek**, J., **Pěchouček**, M. *Multi-agent RRT: sampling-based cooperative pathfinding.* International conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2013), pp. 1263-1264, IFAAMAS, 2013.
9. **Erdem**, E., **Kisa**, D. G., **Öztok**, U., **Schüller**, P. *A General Formal Framework for Pathfinding Problems with Multiple Agents.* Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI 2013), AAAI Press, 2013.

10. **Huang**, R., **Chen**, Y., **Zhang**, W. *A Novel Transition Based Encoding Scheme for Planning as Satisfiability.* Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010), AAAI Press, 2010.

11. **Kautz**, H., **Selman**, B. *Unifying SAT-based and Graph-based Planning.* Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 1999), pp. 318-325, Morgan Kaufmann, 1999.

12. **Kim**, D., **Hirayama**, K., **Park**, G.-K. *Collision Avoidance in Multiple-Ship Situations by Distributed Local Search.* Journal of Advanced Computational Intelligence and Intelligent Informatics (JACIII), Volume 18(5), pp. 839-848, Fujipress, 2014.

13. **KIVA** Systems. *Official web site.* http://www.kivasystems.com/, 2015 [accessed in April 2015].

14. **Kornhauser**, D., **Miller**, G. L., and **Spirakis**, P. G.: *Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications.* Proceedings of the 25th Annual Symposium on Foundations of Computer Science (FOCS 1984), pp. 241-250, IEEE Press, 1984.

15. **Michael**, N., **Fink**, J., **Kumar**, V. *Cooperative manipulation and transportation with aerial robots.* Autonomous Robots, Volume 30(1), pp. 73-86, Springer, 2011.

16. **Ratner**, D., **Warmuth**, M. K.: *Finding a Shortest Solution for the N × N Extension of the 15-PUZZLE Is Intractable.* Proceedings of the 5th National Conference on Artificial Intelligence (AAAI 1986), pp. 168-172, Morgan Kaufmann, 1986.

17. **Rintanen**, J., **Heljanko**, K., and **Niemeӓla**, I. *Planning as satisfiability: parallel plans and algorithms for plan search.* Artificial Intelligence, Volume 170 (12-13), pp. 1031–1080, Elsevier, 2006.

18. **Ryan**, M. R. K. *Graph Decomlocation for Efficient Multi-Robot Path Planning.* Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 2003-2008, IJCAI Conference, 2007.

19. **Ryan**, M. R. K. *Exploiting Subgraph Structure in Multi-Robot Path Planning.* Journal of Artificial Intelligence Research (JAIR), Volume 31, pp. 497-542, AAAI Press, 2008.

20. **Silver**, D.: *Cooperative Pathfinding.* Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2005), pp. 117-122, AAAI Press, 2005.

21. **Sharon**, G., **Stern**, R., **Goldenberg**, M., **Felner**, A.: *The increasing cost tree search for optimal multi-agent pathfinding.* Artificial Intelligence, Volume 195, pp. 470-495, Elsevier, 2013.

22. **Sharon**, G., **Stern**, R., **Felner**, A., **Sturtevant**, N. R.: *Conflict-based search for optimal multi-agent pathfinding.* Artificial Intelligence, Volume 219, pp. 40-66, Elsevier, 2015.

23. **Standley**, T. S., **Korf**, R. E.: *Complete Algorithms for Cooperative Pathfinding Problems.* Proceedings of Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011), pp. 668-673, IJCAI/AAAI Press, 2011.

24. **Sturtevant**, N. R. *Benchmarks for Grid-Based Pathfinding.* IEEE Transactions on Computational Intelligence and AI in Games, Volume 4(2), pp. 144-148, IEEE Press, 2012.

25. **Surynek**, P.: *Towards Optimal Cooperative Path Planning in Hard Setups through Satisfiability Solving.* Proceedings of 12th Pacific Rim International Conference on Artificial Intelligence (PRICAI 2012), pp. 564-576, LNCS 7458, Springer, 2012.

26. **Surynek**, P.: *On Propositional Encodings of Cooperative Path-Finding.* Proceedings of the 24th International Conference on Tools with Artificial Intelligence (ICTAI 2012), pp. 524-531, IEEE Press, 2012.

27. **Surynek**, P.: *Mutex reasoning in cooperative path finding modeled as propositional satisfiability.* Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2013), pp. 4326-4331, IEEE Press, 2013.

28. **Surynek**, P.: *Solving Abstract Cooperative Path-Finding in Densely Populated Environments.* Computational Intelligence (COIN), Volume 30, Issue 2, pp. 402-450, Wiley, 2014.

29. **Surynek**, P.: *Compact Representations of Cooperative Path-Finding as SAT Based on Matchings in Bipartite Graphs.* Proceedings of the 26th International Conference on Tools with Artificial Intelligence (ICTAI 2014), pp. 875-882, IEEE Press, 2014.

30. **Surynek**, P.: *A Simple Approach to Solving Cooperative Path-Finding as Propositional Satisfiability Works Well.* Proceedings of the 13th Pacific Rim International Conference on Artificial Intelligence (PRICAI 2014), pp. 827-833, Lecture Notes in Computer Science, Volume 8862, Springer, 2014.

31. **Surynek**, P.: *Simple Direct Propositional Encoding of Cooperative Path Finding Simplified Yet More.* Proceedings of the 13th Mexican International Conference on Artificial Intelligence (MICAI 2014), LNCS, Volume 8857, pp. 410-425, Springer, 2014.

32. **Surynek**, P. *On the Complexity of Optimal Parallel Cooperative Path-Finding.* Fundamenta Informaticae, Volume 137, Number 4, pp. 517-548, IOS Press, 2015.

33. **Tseitin**, G. S.: *On the complexity of derivation in propositional calculus.* Structures in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics (translated from Russian), pp. 115–125, Steklov Mathematical Institute, 1968.

34. **Wagner**, G., **Choset**, H. *Subdimensional expansion for multirobot path planning.* Artificial Intelligence, Volume 219, pp. 1-24, Elsevier, 2015.

35. **Wehrle**, M., **Rintanen**, J. *Planning as satisfiability with relaxed exist-step plans.* Proceedings of the 20th Australian Joint Conference on Artificial Intelligence, pp. 244–253, Springer, 2007.

36. **West**, D. B.: *Introduction to Graph Theory.* Prentice Hall, 2000.

37. **de Wilde**, B., **ter Mors**, A., **Witteveen**, C.: *Push and rotate: cooperative multi-agent path planning*. Proceedings of the International conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2013), pp. 87-94, IFAAMAS, 2013.

38. **de Wilde**, B., **ter Mors**, A. W., **Witteveen**, C.: *Push and Rotate: a Complete Multi-agent Pathfinding Algorithm.* Journal of Artificial Intelligence Research (JAIR), Volume 51, pp. 443-492, AAAI Press, 2014.

39. **Yu**, J., **LaValle**, S. M. *Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs.* Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI 2013), AAAI Press, 2013.