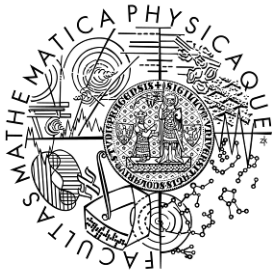

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Monika Daniláková

Artificial Intelligence for the Bang! Game

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Mgr. Martin Pilát, Ph.D.

Study programme: Computer Science

Study branch: General Computer Science

Prague 2017

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In…..... date..... signature of the author

I would like to thank my supervisor, Mgr. Martin Pilát, Ph.D., for his advice and never-ending patience. I would also like to thank my family for their support, and Emiliano Sciarra for creating this wonderful game.

Title: Artificial Intelligence for the Bang! Game

Author: Monika Daniláková

Department / Institute: Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: This work explores artificial intelligence (AI) algorithms for the game Bang!, a Wild West-themed card game created by Italian game designer Emiliano Sciarra. The aim of this work was to design three different AIs for this game and to compare them theoretically and experimentally. First, we analyzed game Bang! with regards to game theory, and researched some of the AI algorithms used in similar games. We then designed three different AIs algorithms and compared their advantages and disadvantages. These three AIs included an AI based on the Monte Carlo Tree Search algorithm, a genetic AI and a hybrid AI using elements of both previous AIs. We also implemented the game itself and the AIs in C#. The implementation makes it easy to add more AIs or client applications in the future, and also to compare and train the AIs. Finally, we experimentally compared the implemented AIs. The genetic AI performed the best, while Monte Carlo Tree Search AI and the hybrid AI were less suitable for this game.

Keywords: multi-player game, artificial intelligence, board card game

Název práce: Umělá inteligence pro hru Bang!

Autor: Monika Daniláková

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí práce: Mgr. Martin Pilát, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Tématem této práce jsou algoritmy umělé inteligence (AI) pro hru Bang!, karetní hru s tématikou Divokého západu, vytvořenou italským vývojářem her Emilianem Sciarrem. Cílem téhle práce bylo navrhnout umělou inteligence pro tuhle hru a teoreticky a experimentálně je porovnat. Nejdřív jsme zanalyzovali hru Bang! s ohledem na teorii her a prozkoumali několik algoritmů umělé inteligence použitých v podobných hrách. Pak jsme navrhli tři různé umělé inteligence a porovnali jejich výhody a nevýhody. Tyhle tři umělé inteligence byli AI používající algoritmus Monte Carlo Tree Search, genetické AI a hybridní AI používající elementy obou předchozích AI. Také jsme naimplementovali samotnou hru a umělé inteligence v jazyce C#. Způsob implementace umožňuje lehce přidávat další umělé inteligence nebo klientské aplikace v budoucnosti, a taky porovnávat a trénovat AI. Nakonec jsme experimentálně porovnali naimplementované umělé inteligence. Evoluční umělá inteligence hrála nejlíp, zatímco Monte Carlo Tree Search AI a hybridní AI byli méně vhodné pro tuto hru.

Klíčová slova: hra více hráčů, umělá inteligence, desková karetní hra

Contents

1 Introduction

Bang![1] is an award-winning card game inspired by the Wild West, created by Italian game designer Emiliano Sciarra. The core of this game is using Bang! cards targeted at other players, which represents shooting at them. The target player can evade the bullet by using a Mancato! card, or he can hide behind a Barile card. However, this is only a small part of the game; there is a great number of other cards, which give the player various special abilities, restore health, extend their shooting range, shoot at several enemies at once, put them in jail, steal their cards, and many others. Each player also has a card which gives them a special ability.

The game also includes a role-playing and team-playing element, which adds another layer of complexity. There are 4 theme-related roles available: the Sheriff, the Renegade, the Outlaws and the Deputies. These roles divide the players into three teams: the Sheriff and his Deputies, the Outlaws, and the lone Renegade. Only one team can win the game. This means that every role has a different objective and calls for a different strategy. Each player's role is kept secret, with the exception of the Sheriff. To distinguish an enemy from an ally, a player must watch his opponent's moves closely to figure out their secret identity. In order to win, the player must change his strategy to suit his role, character, the cards in his hand and other factors.

As far as I could find, there are no existing papers about artificial intelligence for Bang!, which provides a great opportunity to create something new and useful.

This work has several goals. First, we will analyse the properties of game Bang! with regards to game theory. Then we will explore some AIs that have been used in similar games. We will follow with evaluating several possible AI algorithms for this game and choose the three most suitable ones. After that, we will implement both the game and the AIs in C#, in such a way that it would be possible to add new clients and AIs in the future. Finally, we will compare the three AIs implemented.

[1] http://emilianosciarra.net/en/bang/bang

2 Basic game mechanics

The following text explains the basics of game Bang!. This is not the full description of the game's rules. Rather, it is a shortened and simplified explanation of the game's mechanics. The full description of all cards and rules can be found in the official Bang! rules[2] and FAQ[3].

2.1 Terminology

We will need to explain some of the terms we will be using later in this text. Since Bang! uses some specific terminology, it would be difficult to read this text without being familiar with it. We will be using the same terminology as the official game rules whenever possible. Regarding card names, we will be using the original Italian names of the cards. The only exceptions are the role cards, where we will be using their English translations, and the character cards, which are named after fictional characters and require no translation.

Role cards - the cards which are randomly assigned to the players before the game begins, and which determine the player's objective in the game

Character cards - the cards which are randomly assigned to the players before the game begins, and which define player's number of lives and his special ability

Game cards - all Bang! cards which are not role or character cards

Deck - a pile of cards, face down, which players draw cards from

Discard pile - a pile of cards, face up, where players throw away cards to

To draw a card - to draw the top card from the deck

To play or use a card - to use the move written on a card

[2] http://www.dvgiochi.net/bang/bang_rules.pdf

[3] http://www.dvgiochi.net/bang/bang_faq_eng.pdf

To discard a card - to throw a card on the discard pile without playing it

Blue cards - game cards with a blue border. These cards are played by placing them in the discard pile, and their effects are resolved immediately.

Brown cards - game cards with a brown border. These cards are played by placing them on the table in front of a given player, and they have permanent effects.

Table cards - the cards placed on the table directly in front of a player

Hand cards - the cards held in a player's hand

Move - an action done by the player during his turn; either playing a card, discarding a card, using his special move, or ending his turn

Special move - a possible move of a player defined by his character card

Reaction move - an action done by a player, usually outside of his turn, in a reaction to another player’s move

Standard move – a move which is not a special or a reaction move

2.2 Gameplay core

The core of the game is using Bang! cards at other players, which represents shooting at them. The targeted player can duck the shot by using a Mancato! card, or do nothing and lose a life. A player can only shoot at an opponent that is within his shooting distance. The game also contains a great number of other cards, which can give the player special abilities, allow them to attack their opponents in various other ways, restore their health, steal or discard opponent's cards, etc.

2.3 Blue and brown cards

There are two types of game cards available: cards with a brown border and cards with a blue border. In this text, we will call them brown and blue cards, respectively. Brown cards are played by putting them on the top of the discard pile. Their effects are resolved immediately after playing them. Blue cards are played by placing them on the

table in front of the player himself or in front of an opponent. Their effects last until they are removed from the table.

Each game card has three properties: its name, suit and value. The name of the card determines its effects. The suits and values are used in the effects of cards Barile, Dinamite, Prigione and in some special character abilities.

2.4 Turns and phases

Each player’s turn is split into three phases, called Phase 1, Phase 2 and Phase 3. In Phase 1, the player resolves the effects of all Dinamite and Prigione cards on his table. If he is eliminated by the Dinamite, he does not continue his turn. If he remains in jail, he skips to Phase 3. Otherwise, the player draws two cards from the deck. The exceptions to this are characters Black Jack, Jesse Jones, Kit Carlson and Pedro Ramirez, who draw their cards from the deck according to their special ability. In Phase 2, the player can use some of the cards in his hand. Once he discards any card, he enters Phase 3. In Phase 3, the player discards excess cards and ends his turn. The next living player clockwise then starts his turn.

2.5 Roles

At the beginning of the game, each player is randomly assigned a role which defines his objective in the game. The roles are kept secret, with the exception of the Sheriff. Guessing other player's role while hiding one's own plays a great part in the game strategy. Each time a player is killed, his role is revealed. There are four roles available: Sheriff, Deputy, Outlaw and Renegade. There is always exactly one Sheriff and one Renegade in the game. The number of Deputies and Outlaws varies depending on the total number of players.

The Sheriff's objective is to kill all Outlaws and the Renegade. He is the only one whose role is revealed at the beginning of the game. He wins the game if all Outlaws and the Renegade are eliminated while he is still alive.

The task of the Deputies is to protect the Sheriff at all costs. The Deputies win if the Sheriff wins, even if they have been eliminated. This gives them a strong incentive to

protect the Sheriff by eliminating the Outlaws and the Renegade, even at the cost of their own life.

The objective of the Outlaws is to kill the Sheriff. The Outlaws win when the Sheriff is killed while at least one Outlaw or Deputy is still in play. Just like Deputies, the Outlaws all win together, i.e. if one Outlaw wins, they all win.

Finally, the Renegade's task is to first kill all Outlaws and Deputies or wait until they are killed by someone else. When only the Renegade and the Sheriff remain alive, the Renegade must eliminate the Sheriff in a final battle. The Renegade wins if he is the last player in game. However, if the Sheriff is killed while at least one Deputy or Outlaw is still in play, the Outlaws win. Therefore, he has to protect the Sheriff from being killed too soon, and then turn on him at the end of the game.

2.6 Teams

Teams are closely connected to roles. There are three teams in every game: the Sheriff and Deputies, Outlaws, and the Renegade. Each team wins together, i.e. if one member of a team wins, the entire team wins, including its eliminated players. This means that even eliminated players can still win, and gives Deputies and Outlaws incentive to fight for their objective even with their lives. Only one team can win the game.

2.7 Characters

At the beginning of the game, each player is assigned a random character card out of 16 character cards available. The assigned character of each player is publicly known to all players. The characters are inspired by fictional people from the Wild West movies and other stories. The character defines the player's maximum health and the player's special ability. Some special abilities are optional, i.e. the player can choose when and if to use his ability, while some are automatically used whenever possible.

2.8 Distance

The players sit in a circle during the game. A player has a distance of 1 to his immediate left and right living neighbors, a distance of 2 to his neighbor's neighbors, etc.

Eliminated players are not counted in the distance. Whenever a player is eliminated, the distances shrink. Initially, each player can only shoot to the distance of 1, but there are blue cards which can extend this range or modify the distance itself. This distance is not symmetrical; the distance at which player A sees player B might be different from the distance at which player B sees player A.

2.9 Health

Bang! uses a simple health system. Each player has a maximum health of 3-5 lives, depending on their character and role. Each player starts with the number of lives equal to the number of bullets depicted on their character card. Most character cards have 4 bullets, with only 2 characters with 3 bullets. The Sheriff starts with one extra life. A player's current health also determines the maximum numbers of cards in his hand at the end of his turn. A player is not allowed to end his turn while the number of cards in his hand exceeds the number of his remaining lives. In this case, he has to dispose of some cards by playing or discarding them before ending his turn.

2.10 Damage

Almost all game cards either deal no damage or a take away 1 life. The only exception is the Dinamite card, which takes away 3 lives, often eliminating the affected player. A player is eliminated if he loses his last life and does not immediately use a Birra card.

2.11 Endgame

The game ends when one of two scenarios happen. First, the game is over when the Sheriff is killed. The Renegade wins if he is the only surviving player. Otherwise, the Outlaws win, including eliminated Outlaws. Second, the game also ends if all Outlaws and the Renegade are killed while the Sheriff is still alive. In this case, the Sheriff and all of his Deputies win the game, including eliminated Deputies.

3 Game analysis

Bang! is a game with imperfect information. A game with imperfect information is a game where a player may not know the full game state (Guillermo, 2013). In Bang!, a player often does not know the roles of some of his opponents, the cards in their hands, and the order of the cards in the deck.

Bang! is a game with incomplete information. A game with incomplete information is a game where a player may not know the full game state or the full outcome of some previous actions (Frank & Basin, 1998). In addition to the unknown game state information mentioned above, the player usually does not know which card an opponent draws from the deck or steals from another opponent's hand.

Bang! is a stochastic game, which means that it includes some random elements. This includes the randomly assigned role cards, character cards, and the randomly shuffled deck.

Bang! is an infinite game, since the rules theoretically allow for it to continue indefinitely (Guillermo, 2013). At least one player has to lose all of his health for the game to end, and there are plenty of moves which have no effect on health. Also, a player does not have to make any moves during his turn. In any of his turns, a player is allowed to merely discard his excess cards and end his turn.

Bang! is a sequential, turn-based game. Only one player can make a move at a time.

Bang! is a non-zero-sum game. A zero-sum game is a game in which a player's gain automatically causes an equal loss to the opponent (Millington & Funge, 2016). This is not true in Bang!. Firstly, a Bang! player gaining a card does not affect any of the opponent's cards, since cards are usually drawn from the deck. The only exception is the Panico! card, but this is an exception rather than the rule. Secondly, changes to one player's health do not affect the health of any other players.

4 Related work

4.1 Artificial intelligence in similar games

Before we can select the three most suitable AIs to implement, we will need to research some AIs that have been succesfully used in similar games. We will limit our research to sequential games for two or more players, preferably games with imperfect and incomplete information or stochastic games.

In 2016, program AlphaGo developed by Google DeepMind achieved a great success in game Go. Go is a Chinese turn-based, two-person, zero-sum, deterministic board game of perfect information with an extremely large search space. It is considered to be one of the most complex board games (Chaslot G. M.-B., 2010). AlphaGo used a mix of Monte Carlo simulation and neural networks. It achieved 99.8% victory rate against other Go programs, and defeated a human professional player on a full-sized Go board for the first time in history (D. Silver, 2016).

In 2010, a Monte Carlo Tree Search AI algorithm was successfully used in game Kriegspiel. Kriegspiel is a modification of chess where both players can see only their own pieces. There is a third player added, called the referee, who can see all pieces and who informs the players whether they can make a given move. This makes Kriegspiel a game of incomplete and imperfect information, a property which it shares with Bang!. The Monte Carlo Tree Search AI has been able to beat a minimax Kriegspiel AI and to play well against human players (Ciancarini & Favini, 2010).

4.2 Existing Bang! implementations

My implementation of Bang! is not the first virtual implementation of this game.

There is an official Bang! video game[4] available for iOS, Samsung, Windows and other platforms. The source code for this program is, of course, proprietary.

[4] http://www.bangvideogame.com/

There is also an unofficial online version[5] of Bang! available under the name "Kraplow!", made by software developer Christopher Gordon Carr. Its source code is available online on Github[6]. This implementation includes multiplayer, single player, AI, chat, game log and Ajax polling. However, since this code lacks documentation and comments of any kind, it would be very difficult to use and expand it in order to include my AIs, AI training and testing. I have decided that it would be much easier and better to create my own implementation of this game.

4.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a best-first tree search algorithm which explores the search space in a randomized way (Chaslot G. M.-B., 2010).

One step of Monte Carlo Tree Search algorithm consists of four main phases: selection, expansion, simulation and backpropagation. During selection, we start at the root of the tree and traverse the tree until we arrive at the desired node. We then select a child of this node which has not been yet added to the tree, and add this child node during the next step, expansion. This is followed by the simulation phase, where the game is played out from this child node until the end. The MCTS step finishes with the final phase, backpropagation, where the result of the played game is propagated backwards, all the way to the root. The MCTS steps are repeated as many times as the time limit allows. Once he the time limit runs out, we select the child node of the root with the largest number of playouts, rather than victories (Chaslot G. M.-B., 2010).

There are several algorithms available for selecting nodes in the selection phase. In this work, we will use the UCT (Upper Confidence bounds applied to Trees) algorithm. UCT selects child node *k* of the current node *p* according to this formula:

$$k \in argmax_{i \in I} \left(v_i + C * \sqrt{\frac{\ln n_p}{n_i}} \right)$$

[5] http://chriscarr.name/westerncardgame/

[6] https://github.com/ccarrster/kraplow

where I is the set of children of node p, v_i is the value of the node i, n_i is the visit count of i, n_p is the visit count of $_p$, and C is a constant which needs to be determined by trial and error (Chaslot G. M.-B., 2010).

4.4 Evolution strategies

Evolution strategies (ES), also called evolutionary strategies or evolutionary programming, are algorithms inspired by evolution in living organisms (Hansen, Arnold, & Auger, 2015).

A vital part of an ES algorithm is a population of vectors, called individuals, which are analogous to living organisms. Each individual is a vector of values, called traits, analogous to DNA of the orgasnisms. Every individual also includes a fitness function, which determines how good or favorable the individual is based on the specific environment. An ES algorithm consists of three stages: mating selection, mutation and environmental selection. These stages are repeated in a cycle as long as time allows. (Hansen, Arnold, & Auger, 2015).

Each iteration of the ES algorithm, also called a generation, begins with mating selection. In this stage, one or several original individuals are selected (called parents), and new individuals are created (called offspring) by duplication or recombination of the selected parents. Recombination includes combining traits from multiple parents into a single new indvidual, while duplication simply creates an identical clone of the parent.

In the next step, offspring undergo mutation. This is analogous to DNA mutation in living organisms. Like DNA mutation, ES mutation makes small, random and unbiased changes to the individuals. Typically, all values of the individual are mutated.

In the last step, n individuals are selected by environmental selection based on their fitness function. The n fittest individuals are selected, the rest are discarded and the population size returns to the original size. (Hansen, Arnold, & Auger, 2015)

5 Rule modifications

Since there are some inherent differences between human and AI players, we need to make several adjustments to the game rules. Some of these changes simplify gameplay, others deal with players closing the program during a game and some changes close rule loopholes that could allow for the game to continue indefinitely.

- While the original game has no time limit or any other duration limit, we will add a limit on the number of rounds allowed per game. While human players are unlikely to play a Bang! game for an unreasonably long time due to boredom, AI players have no such reservations. This number can be adjusted in the client application. If this round limit is exceeded, all players lose the game.
- The special ability of Suzy Lafayette allows her to draw one card whenever she has no cards in her hand. This could potentially lead to her discarding and drawing cards indefinitely. Also, the round limit rule does not prevent this. That is why we need to add a rule that when she discards her last card during her turn and receives a card according to her special ability, she cannot discard another card during that turn.
- The effect of the Barile card will be used automatically whenever possible. This is a slight change from the original rules, where using this card is optional. However, since using this card offers a 25% chance of deflecting a Bang! card and has no downsides, it is optimal to use Barile whenever possible.
- Lucky Duke's special ability will be used whenever possible. This is a slight change from the original rules, where using this card is optional. The reasoning is similar to the Barile scenario.
- If all players leave before the game before it is over, all players lose the game.
- If a player leaves the game before being eliminated, he is killed in the game and the game continues with the remaining players.

6 Implementation

6.1 Used frameworks and programs

The server program and the client program are implemented in C# using the .NET 3.5 framework. This framework was chosen primarily because of its extensive 2D drawing, JSON (JavaScript Object Notation) serialization and TCP (Transmission Control Protocol) communication libraries, all of which were used in the implementation. The implementation was done in Microsoft Visual Studio Ultimate 2013[7]. All graphical work was done in GIMP[8] (GNU Image Manipulation Program).

6.2 Card graphics

The client application uses the original card art of the game. Mr. Sciarra, the creator of Bang!, has kindly allowed me to use it.

6.3 Basic framework

The implementation of Bang! consists of two applications: the server application and the client application. Their code is stored separately in their respective Visual Studio 2013 solutions. Detailed information about the implementation is available in the user's guide, the programmer's guide and the communication documentation, which can be found on the attached CD.

6.3.1 Client

The client application allows human and AI players to participate in the game. It also allows easy testing and training of AIs. It uses Windows Forms UI to display current game information to the player and to collect user input. Multiple instances of the client can run simultaneously on the same computer.

[7] https://www.visualstudio.com/en-us/news/releasenotes/vs2013-rtm-vs

[8] https://www.gimp.org/

6.3.2 Server

The server manages all running and waiting games, stores all information of all running games, and directs the flow of each game. The server can manage many games at once. The number of possible games running at the same time is only limited by the server hardware. The server also prompts the clients to make a move when needed, checks the validity of every attempted move, and sends messages to clients to make sure all clients always have the most current game information.

The server is designed in such a way that it is possible to add new client applications in the future, which could even be implemented in a different programming language or made by a different software developer. An interesting use of this feature would be making an AI coding competition for software developers.

The server checks the validity of any move an AI player or a client application attempts to make. If the move is legal, the server performs the move and sends the updated game information to all other players. If the move is illegal, the client is notified and prompted to try a different move. This move checking design makes it impossible for an AI or a client application to cheat by making an illegal move. It also protects the game integrity from possible bugs in an AI or in a client application made by other developers. This feature makes it easy to integrate new AIs or new client applications in the future.

During a game, only the server has access to the full game information. This prevents any AI or any client application from cheating by accessing information that should be hidden from them, such as cards in other player's hands, another player's role, or the order of the cards in the drawing pile. The client and the AI can only access the information that would be available to a player in a real-world game of Bang!.

6.3.3 Client-Server communication

When the client is launched, it will attempt to establish a TCP connection to the server. This connection will remain open until either the client or the server closes. Every time the server receives the connection from a new client, it will launch a new thread dedicated to listening for messages from that specific client. The only communication happens between the server and the client. The clients do not communicate with each other.

Every message is an instance of the BangMsg object. This object is serialized, sent to the recipient and deserialized back into a BangMsg object. The format of messages is described in detail in the client-server communication documentation.

6.4 Server-client communication

The server communicates with the clients via TCP sockets using native .NET libraries `System.Net` and `System.Net.Sockets`. TCP was chosen because of its portability, popularity, ease of use and no need for third-party libraries.

6.5 Serialization

The client communicates with the server by sending JSON messages over a TCP connection. The JSON messages are created by serializing instances of C# objects, and are later deserialized back into objects. There are several reasons for choosing JSON over other similar technologies.

The first reason is its portability. Since JSON messages are simply strings of plain text, the only adjustment needed for communication between different devices or operating systems is setting the correct encoding and line ending standard, which is trivial.

Second, the wide-spread use of JSON has led to the creation of publicly available JSON serialization libraries for many popular programming languages. In case new client applications were added at some point in the future, the author would be able to choose from a large variety of programming languages.

Another great reason is JSON's efficiency, especially when compared to XML (Extensible Markup Language). XML has an inherent efficiency disadvantage, as the name of each item is always repeated twice in the message, in the item's opening and closing tags. JSON does not suffer from this disadvantage, since it uses brackets instead of tags to delimit items. This makes JSON messages shorter than their XML equivalent, accelerating communication and easing the workload of the server.

Finally, the use of JSON allows the server and client to be implemented in C# without the need for any third-party libraries. Since version 3.5, .NET contains the native

library `System.Runtime.Serialization.Json` and its class `DataContractJsonSerializer`, which was used in the implementation.

7 Artificial Intelligence

7.1 Move types

There are three basic types of moves a Bang! player can make: reaction moves, start turn reaction moves, and normal moves. Each AI (artificial intelligence) must be able to make moves of all of these types.

A reaction move is a move made outside of the player's turn, based on another player's actions. Reactions are needed after using cards Emporio, Bang!, Indiani!, Duello and Gatling, and when the player is losing his last life. In all cases except for the Emporio, a player can react by using a card or by doing nothing. However, a "do nothing" reaction to Bang!, Indiani!, Duello and Gatling invariably causes a loss of one life. A "do nothing" reaction to losing the last life causes the player to be eliminated. This means that doing nothing is usually unfavorable.

A start turn reaction move is the use of special ability of Jesse Jones, Pedro Ramirez and Kit Carlson in the first phase of their turn. When Jesse Jones draws his two cards in Phase 1 of his turn, he may choose to draw his first card from the hand of any opponent or from the deck. Pedro Ramirez is allowed to draw his first card from the discard pile or from the deck. Kit Carlson draws three cards instead of two, and puts one back in the deck, face down. The remaining characters always simply draw 2 cards in the first phase of their turn, which happens automatically and requires no decision on their part. This means that only characters Jesse Jones, Pedro Ramirez and Kit Carlson are able to make start turn reaction moves.

A normal move is a Bang! move which is not a reaction move or a start turn reaction move. Normal moves can be split into five further groups: targeted moves, untargeted moves, special moves, ending the turn and discarding a card. Targeted moves include using cards Bang!, Mancato! used as Bang!, Prigione, Duello, Panico! and Cat Balou. Untargeted moves include using any other card. A special move is the use of Sid Ketchum's special ability. Ending the turn means the player does not want to make any other moves in his turn, and the next living player clockwise starts his turn. Discarding a card means placing a card on top of the discard pile without using its effects.

7.2 Algorithms used by multiple AIs

There are some algorithms which are used by multiple implemented AIs. These algorithms include tracking of knows cards in the opponent's hands, tracking the opponent's actions and guessing the opponent's roles.

7.2.1 Tracking cards in opponent's hands

During a game, a player usually does not know the values of cards of opponent's hands. However, some of these hand cards may become known during the game. For example, this may happen after an opponent uses Emporio, Panico!, or some of the special abilities. When a player uses Emporio, several cards are placed on the table, face up, and every living player picks one. When an opponent uses Panico! to steal a table card from another player, he puts the stolen card, which is known to everyone, in his hand. When an opponent uses Panico! to steal a table card from the AI's hand, the card is also known, since the AI knows which cards are in its hand.

The algorithm tracking these hand cards is trivial, so it will not be explained further. Card tracking is especially useful for AIs which search the game tree, as it makes their playouts closer to reality.

7.2.2 Tracking of opponent's actions

Tracking of opponent's actions is essential for guessing their roles. At all times, the given AI will keep a table T of all targeted attacks made in the current game. Table T will be a *n*x*n* matrix, where n is the number of players at the beginning of the game. Each item T_{ij} will keep information about attacks made by player P_i targeted at player P_j. All items of the matrix will be initiated to zero at the beginning of each game. Whenever an opponent makes a move which suggests his role, the relevant counters will be increased by a given number of points.

Note that there are moves which do not offer much help in determining an opponent's role as their effects are "untargeted", i.e. either directed at the player himself, or at all living opponents. For example, this would include using cards Saloon, which adds one

life for everyone, or Wells Fargo, which lets the player draw three extra cards. In some cases, it could possible to use an untargeted card in a somewhat targeted fashion, e.g. using Saloon when only some players can benefit from it. However, it would be difficult to distinguish whether this was intentional or not. Therefore, these untargeted cards will have no effect on the counters. Only the targeted attacks will affect the counters. These targeted attacks are listed below.

7.2.2.1 Targeted attack types

Targeted moves include using cards Bang!, Mancato used as Bang! (Calamity Janet's special ability), Duello, Panico!, Cat Balou and Prigione. We will define four basic types of targeted moves: strong attacks, weak attacks, assistance and neutral moves.

A "strong attack" is a targeted move which can directly lower a specific opponent's health. This includes Bang!, Mancato used as Bang!, and Duello. It does not matter if this actually lowers the target player's health, or if the target manages to defend himself. Only the attacker's intentions matter here, not the results. Since this attack indicates a very strong intention to eliminate the target, this attack will add 3 points to the relevant counters.

A "weak attack" is an attack which cannot directly lower the target's health, but will inconvenience the target in a different way. This includes Panico!, Cat Balou and Prigione, but only if the Panico!, Cat Balou is not used on a Dinamite or Prigione on the target's table. The weak attack will add 1 point to the relevant counters.

"Assistance" is a move that directly helps a specific player. It only includes using Panico! or a Cat Balou used to remove a Prigione card from another player's table cards, protecting the target from its effects. Since assistance to a given player indicates a very strong intention to help this player, it will subtract 3 points from the relevant counters.

A "neutral" targeted move includes using Panico! or Cat Balou to remove a Dinamite card from another player's table cards. Despite being targeted, these two moves do not give much information about the player's intentions. It would be difficult to decide if the player intended to protect the opponent who currently has the Dinamite, to protect a different player before the Dinamite could travel to their table, or simply to protect himself. For this reason, the neutral targeted moves will have no effect on the counters.

7.2.2.2 Move tracking algorithm

Here is the pseudocode describing the move tracking algorithm:

```
At the beginning of the game:
      N = the number of players
      Table T = a table of size NxN, initialized to 0
      List P = list of all players

Each time player Pi makes a targeted move M at player Pj:
      If move M is a strong attack
            T[i,j] += 3
      Else if move M is a weak attack
            T[i,j] += 1
      Else if move M is an assistance move
            T[i,j] -= 3
```

7.2.3 Guessing the opponent's roles

Role guessing is an integral part if any Bang! strategy, since it is a team-playing game and the knowledge of other player's roles is essential for distinguishing enemies from allies.

First, the algorithm will try to determine an opponent's role by eliminating all other possibilities. Every player knows the list of all roles used in the game, although they do not know their distribution. The list of all roles is given in the rules, and depends only on the number of players at the start of the game. Guessing roles by elimination becomes more effective as the game continues, since the roles of players are revealed as they are killed. The elimination algorithm is very straightforward. It tracks the roles of the Sheriff, the player's own role, the roles of all eliminated players and the roles with unknown owners. If all remaining roles with unknown owners are identical, all players with unknown roles are simply assigned this last remaining role.

If there are still players with unknown roles after the elimination, the algorithm will guess the opponent's roles based on their previous actions, which are recorded by the move tracking algorithm described above. The algorithm will order the players with

unknown roles based on their previous attacks on the Sheriff. Then it will assign Outlaw role to those who attacked the Sheriff the most, Deputy roles to those who attacked him the least, and the final remaining player will be assigned the Renegade role. This second part of the algorithm is based on the assumption that Outlaws should attack the Sheriff the most and the Deputies should attack him the least. The Renegade should be somewhere in the middle, since he needs to weaken the Sheriff, but not eliminate him. This assumption may not always be accurate. For example, it may be wrong if an opponent is actively trying to hide his true role. However, it is the best possible guess based on the information available to the AI. This algorithm only uses the attacks on the Sheriff, since his role is known to everyone. Of course, attacks on other players could also give some information about the attacker's intentions. However, this information is less reliable, since we cannot always be sure that the attacker guessed the target's role correctly. The AI does not know how good its opponents are at guessing roles. Using only attacks on the Sheriff prevents one player's incorrect guess from confusing everybody else.

Pseudocode

GuessRoles(C, T, N)

Parameters

- C – all known game information
- T – table from the opponent move tracking algorithm
- N – the number of players (including eliminated players)

Pseudocode

```
List K = list of length N
For i = 0 .. N – 1
      If i = sheriffs ID
            K[i] = Sheriff
      Else if i = our player's ID
            K[i] = our player's role
      Else if player Pi is dead
            K[i] = role of player Pi
      Else
            K[i] = unknown
```

```
If list K contains no unknown values
      Return list K
List U = list of all unassigned roles (roles in the game that are
not in list K)
If all elements of list U are identical
      Replace all “unknown” values in list K with value U[0]
      Return list K
L = list of all players with unknown roles
Sort list L by “anti-Sheriff points” of its players ascending, where
anti-sheriff points of player Pi are equal to T[I, sheriff’s ID];
break ties randomly
While list U contains at least 1 element:
      If list U contains 1 element:
            P = the only player in list L
            i = ID of player P
            K[i] = the last item in list U
            Return list K
      If list U contains at least one “Outlaw” item
            P = last player in list L
            i = ID of player P
            K[i] = Outlaw
            Remove player P from list L
            Remove one “Outlaw” item from list U
      If list U contains at least one “Deputy” item
            P = first player in list L
            i = ID of player P
            K[i] = Deputy
            Remove player P from list L
            Remove one “Outlaw” item from list U
```

7.3 Implemented AIs

We have decided to design and implement five different AIs. Two of the AIs, Random AI and Random AI with roles are very simple and relatively weak, and they serve as a baseline for testing the more advanced AIs. The next three AIs are much more advanced, and their aim is to play comparably to a human player. This includes Monte Carlo Tree Search AI, Evolution AI and Hybrid AI.

7.4 Random AI

7.4.1 Overview

Random AI (RAI) is used mainly for evaluating the quality of other AIs. It allows us to rate the more advanced AIs based on how much better they play than RAI. Random AI makes its moves almost randomly, with only two simple heuristics. Firstly, it prefers using cards to discarding them or to ending their turn. Secondly, it prefers reacting to an attack to not reacting.

7.4.2 Reaction moves

If RAI is reacting to Emporio, it chooses its reaction at random. If it is reacting to anything else, it has two choices: to react by using a card or do nothing. Since making a “do nothing” reaction is almost always unfavorable, RAI will only choose to “do nothing” if it has no other choice.

Pseudocode

```
List M = all valid reaction moves
If the AI is reacting to Emporio
      Return a random move from list M
Else
      If list M contains a “do nothing” move, remove the “do
      nothing” move
      If list M contains at least 1 move
            Return a random move from list M
      Else
            Return a “do nothing” move
```

7.4.3 Start turn reaction moves

RAI will simply choose a random valid move.

Pseudocode

```
List M = all valid start turn reaction moves
Return a random move from list M
```

7.4.4 Normal moves

Making a random normal move is a bit more complicated than making a random reaction or start reaction move. The AI cannot simply make a list of all valid moves and randomly choose one item. Firstly, this naïve implementation would strongly favor targeted moves. If a targeted card has *n* possible targets, it would appear in this list *n* times, and therefore be *n* times more likely to be chosen than an untargeted card. Secondly, the naïve implementation would also favor discarding cards, since every card can be discarded, but not every card can be used. Since discarding a card is generally very unfavorable, this would make the AI play very poorly.

Instead, the RAI uses the following implementation: When making a normal move, the RAI will first randomly choose whether to use a card, discard a card, end its turn or make a special move (if possible). However, not all of these choices are equally favorable, and therefore need to have different probabilities of being chosen.

Discarding a card is the least favorable for obvious reasons, so it needs to have the lowest probability. Ending the turn needs to have a lower probability than using a card, since players in Bang! generally use more than one card per turn. Special move is Sid Ketchum's special ability, which allows him to discard 2 cards and gain 1 life in return. The special move makes him run out of cards quickly, and therefore should have a lower probability than using a card.

This means that each option needs to have a specific weight. A weight of *n* means that the option will be *n* times more likely to be chosen than it would have been otherwise. Using a card will have weight 4, ending the turn will have weight 2, special move will have weight 2 and discarding a card will have weight 1. These numbers were selected based on the knowledge of the game, and also tuned by trial and error.

If the AI chooses to discard a card, the card will be then chosen at random. If it chooses to use a card, the card will be again chosen at random, with a random target. If the AI chooses to end its turn, it does not have to make any more choices. Finally, if it chooses to use the special move, it will choose its two cards at random.

Pseudocode

```
List L = list of all valid moves
List P = empty list
If list L contains at least one discard move
      Add "Discard" to list P
If list L contains end turn move
      Add "End turn" to list P twice
If list L contains at least one special move
      Add "Special" to list P twice
If list L contains at least one use card move
      Add "Use" to list P four times
M = a random element from list P
If M = "Discard"
      Discard a random card
Else if M = "End turn"
      End the turn
Else if M = "Special"
      Make a special move using two random cards
Else
      List T = a list of all cards which can be used
Card C = a random item from list T
      If card C is untargeted
            Make an untargeted move with card C
      Else if card C is targeted
            Make a targeted move with card C at a random target
```

7.5 Random AI with roles

7.5.1 Overview

The Random AI with roles (RAIR) is very similar to the Random AI (RAI). However, RAIR also uses the available role information in its decision making process. RAIR is also used mostly for evaluating the quality of other AIs. The main advantage of RAIR over RAI is that RAIR is closer in behavior to humans or more advanced AI players.

RAIR will avoid attacking allies and helping enemies. First, it guesses the roles of all opponents using the role guessing algorithm. Then it will take the list of all valid possible moves and removes all targeted moves which would harm an ally or help an enemy. Finally, it chooses a move from this modified list using RAI algorithm.

There are three teams in each game: the Sheriff and the Deputies, the Outlaws, and the Renegade. Player's allies are usually all of his teammates and his enemies are all opponents from other teams. The only exception is the Renegade, who has to keep the Sheriff alive until only he and the Sheriff remain, and then turn on the Sheriff. That means that if there are more than two players alive, the Renegade's ally will be the Sheriff and his enemies will be the Deputies and Renegades. If there are only two players alive, the Sheriff will become the Renegade's enemy.

7.5.2 Reaction moves

The reaction move selection algorithm is identical to that of RAI, since reaction moves never have a target.

7.5.3 Start turn reaction moves

The start turn reaction selection algorithm is almost identical to that of RAI, with the exception of character Jesse Jones. His special ability is that in Phase 1 of his turn, he can take his first card from the deck or from the hand of any other player. Obviously, a RAIR player should avoid stealing cards from his allies. The start turn reactions of the other two characters, Pedro Ramirez and Kit Carlson, are not targeted at opponents and are chosen by RAI algorithm.

Pseudocode

Parameters:

- G – all available game information
- Table T from the move tracking algorithm

Pseudocode:

```
List V = list of all valid moves
If the player's character is Jesse Jones
      List V2 = an empty list
      List R = GuessRoles(G, T)
      For each move M in list V
            If the target of move M is not an ally (based on list R)
                  Add M to list V2
      V = V2
Return a random item from list V
```

7.5.4 Normal moves

RAIR takes the list of all possible moves and removes all targeted moves which attack allies or help enemies. Then it chooses a move from this modified list using RAI algorithm.

Pseudocode

Parameters:

- G – all available game information
- Table T from the move tracking algorithm

Pseudocode:

```
List R = GuessRoles(G, T)
List V = list of all valid moves
List V2 = an empty list
For each move M in list V
      If M is targeted
            If move M is a strong or a weak attack
                  If the target of move M is not an ally (based on
                  list R)
                        Add M to list V2
```

```
            Else if move M is neutral
                  Add M to list V2
      If move M is assistance
                  If the target of move M is an ally (based on list
                  R)
                        Add M to list V2
      Else if M is targeted
            Add M to list V2
Return a move from list V2 using RAI
```

7.6 Monte Carlo Tree Search AI

7.6.1 Overview

Monte Carlo Tree Search AI (MCAI) relies primarily on Monte Carlo Tree Search (MCTS) algorithm, which was described in Chapter 4. MCAI uses the same algorithm for normal moves, reaction moves and start turn reaction moves.

First, MCAI picks *n* different random permutations of all unknown game cards. Unknown game cards include the cards in the deck and the all cards the opponent's hands which are not revealed by the card tracking algorithm. This removes all random elements from the following simulation, reducing the width of its move tree and simplifying the algorithm. The value of *n* needs to be determined by trial and error, as it depends on the speed of the specific machine and the maximum time limit allowed for the computation. A good starting value is 8, which was used during performance testing of this AI.

Then, MCAI creates *n* copies of all known information about the game and uses the *n* permutations to fill out the missing game information. This results in *n* different sets of complete game information. We will call these sets seeds. For each seed, the AI then searches its move tree using MCTS.

7.6.2 Parallelization

Since MCTS is computationally intensive, it should be implemented in a parallel way in order to maximize its performance. There are three main parallelization metods for MCAI: leaf parallelization, root parallelization, and tree parallelization (Chaslot, Winands, & van den Herik).

We will use tree parallelization for MCAI. This parallelization method is lock-free and easy to implement. Instead of assigning all *n* trees to the same thread, we will simply evenly distribute the trees between the threads. Assuming we have *n* trees and *m* threads, we will assign every tree T_i to thread $P_{i \bmod m}$.

7.6.3 Tree node

Each node of the tree will contain the following information:

- Game – the complete information about the game state of the node
- Parent – parent node
- MoveMaker – player whose move resulted in this node
- Children – child nodes
- Playouts – the number of playouts
- Wins – the number of playouts that resulted in victory for player M

7.6.4 Choosing a move

The following pseudocode describes a single-threaded implementation in order to make it easier to read and understand. The conversion to multi-threaded implementation is trivial; we would simply split the seeds between the threads instead of assigning all seeds to the same thread.

7.6.4.1 Main function

Parameters:

- Time limit T – the time limit in which the AI must make its decision
- List A – list of all possible moves the AI can currently make
- N – number of permutations to be searched
- List U – list of all unknown game cards
- G – all game information known to the AI

Pseudocode:

```
List S = empty list
For i = 0 .. n - 1
      P = a random permutation of list U
      C = a copy of G
      Distribute the cards from list P to game state C so that every
      opponent has the correct number of cards in their hand
      N = node with game state C and no parent
      Add N to list S
While time limit T has not passed:
      Select a random element Si from list S
      C = a deep copy of Si
      GuessRoles(C)
      SelectNode(C)
      Expand(C)
List X = list with equal length to list A
For each move Ai from list A
      Xi = 0
```

```
      For each seed Sj from list S
            N = the child node of seed Sj created by move Ai
            Xi = Xi + N.Wins / N.Playouts
      Xk = the item of list X with the highest value; break ties
      with a random choice
Return move Ak
```

7.6.4.2 Helper functions

SelectNode(node R)

```
Node N = R
While all possible children of node N exist and have at least 1
playout
      List C = all children of node N
      N = the element from list C with the highest UCT value
Return N
```

Expand(node N)

```
If possible children of node N are reaction moves
      If node N has no children
            C = a list of all possible children of node N, except
            children made by "do nothing" move
      Else
            C = a list of all non-existent children of node N
Else if possible children of node N are start turn reaction moves
      C = a list of all non-existent children of node N
Else
      If node N has no children
            C = a list of all possible children of node N, except
            children made by discard move
      Else
            C = a list of all non-existent children of node N
For each item S of list C
      Add S to children of node N
      W: = Playout(S)
      BackPropagate(S, W)
```

BackPropagate(Node N, list W)

```
Do
      N.Playouts++
If list W contains player N.MoveMaker
            N.Wins++
      N = N.Parent
While node N has a parent
```

Playout pseudocode

Parameters:

- S – game situation

Returns:

- W – the list of winning players

Pseudocode

```
While the game is not over
      If there is at least one pending reaction request
            Select one reaction request R
            M = a reaction move selected by RAI based on request R
      Else there is a pending start turn reaction request
            M = a start turn reaction move selected by RAI based on
            the request
      Else
            M = a normal move selected by RAI
      Make move M and change situation S accordingly
Return the list of winning players
```

7.6.5 Advantages and disadvantages

MCAI has the great advantage of being able to plan ahead. Also, it requires no training or knowledge of strategy for the particular game, although some knowledge could be beneficial for heuristics. This makes this AI relatively easy to implement. Another advantage is its ability to be easily implemented in a multi-threaded way. Finally, it implicitly takes every part of the game information into account: the roles, the characters, the known cards, the playing order, distances between players, etc.

Its main disadvantages include its high demands on computing power, and its potential to play poorly if the number of performed simulations is insufficient. The number of simulations may be too low if the time limit for calculation is low, or the machine has insufficient computational power. The quality of its decisions also greatly depends on how close the searched random permutations are to the actual game situation. This means that the quality of this AI can fluctuate randomly between games and also within the same game. Finally, since MCAI does not makes its decisions instantaneously, human players could find it frustrating to play against.

7.7 Evolution AI

7.7.1 Overview

Evolution AI (EAI) uses an evolution algorithm, which was described in Chapter 4. EAI explores the possibility of playing Bang! without searching the move tree. It uses a mathematical formula to score game states based on how favorable they are to the player. Each time EAI needs to make a move, it simulates all possible moves and notes the score of each resulting game situation. The score depends on the health, hand cards and table cards of the player, his allies and his enemies. Then, EAI selects the move which resulted in the situation with the most favorable score. If there are multiple moves which result in the best score, it randomly chooses one of them.

This AI is trained using the client application. The number of matches and individuals per generation can be changed by the user and should be determined by trial and error. The population has only one parent, and *n* offspring are generated in each generation. At the end of each generation, the most successful individual, let us call it S, is selected and replaces the original individual. In the notation used in evolution strategies, such an algorithm would be called *(1 + n)-ES*. (Hansen, Arnold, & Auger, 2015).

7.7.2 Individual

EAI uses individuals which play against each other and evolve. Each individual stores information used for picking moves.

The individual consists of:

- Renegade multipliers
- Outlaw multipliers
- Sheriff multipliers
- Deputy multipliers

The multipliers for a given role consist of:

- Health multipliers:
    - Player's health multiplier
    - Renegade health multiplier
    - Outlaw health multiplier
    - Sheriff health multiplier
    - Deputy health multiplier

- Player's multipliers:
    - Volcanic multiplier
    - Barile multiplier
    - Maximum shooting distance multiplier
    - Mustang multiplier
    - Mirino multiplier
    - Hand cards count multiplier
- Ally multipliers:
    - Volcanic multiplier
    - Barile multiplier
    - Maximum shooting distance multiplier
    - Mustang multiplier
    - Mirino multiplier
    - Hand cards multiplier
    - Prigione multiplier
- Enemy multipliers:
    - Volcanic multiplier
    - Barile multiplier
    - Maximum shooting distance multiplier
    - Mustang multiplier
    - Mirino multiplier
    - Hand cards count multiplier
    - Prigione multiplier

7.7.3 Default individual

Initial individual data can be found in the electronic attachment in file "Application\Default AI individuals\Evolution AI.txt". I chose its values based on my knowledge of the game.

7.7.4 Game state score calculation

The following pseudocode explains the game state score calculation.

Parameters:

- Game state G
- N – player count
- A – the AI's player

Pseudocode:

```
score = 0
Guess each player's role
for i = 0..N - 1
```

```
      H = health of player Pi
      If player Pi = A
            score += player's own health multiplier * H
      Else
            If role of player Pi = deputy
                  score += deputy health multiplier * H
            If role of player Pi = outlaw
                  score += outlaw health multiplier * H
            If role of player Pi = renegade
                  score += renegade health multiplier * H
            If role of player Pi = sheriff
                  score += sheriff health multiplier * H

for i = 0..N - 1
      If player Pi = A
            If player Pi has Volcanic on his table or his character
            is Willy the Kid
                  score += player's Volcanic multiplier
            If player Pi has Barile on his table
                  score += player's Barile multiplier
            If player Pi has Mirino on his table
                  score += player's Mirino multiplier
            If player Pi has Mustang on his table
                  score += player's Mustang multiplier
            score += max shooting distance of player Pi * player's
            max shooting distance multiplier
            score += number of cards in player Pi's hand * player's
            hand cards multiplier
      Else if player Pi is an ally
            If player Pi has Volcanic on his table or his character
            is Willy the Kid
                  score += ally Volcanic multiplier
            If player Pi has Barile on his table
                  score += ally Barile multiplier
            If player Pi has Mirino on his table
                  score += ally Mirino multiplier
            If player Pi has Mustang on his table
                  score += ally Mustang multiplier
            score += max shooting distance of player Pi * ally max
            shooting distance multiplier
```

```
            score += number of cards in player Pi's hand * ally hand
            cards multiplier
            If player Pi has Prigione on his table
                  score += ally Prigione multiplier
      Else if player Pi is an enemy
            If player Pi has Volcanic on his table or his character
            is Willy the Kid
                  score += enemy Volcanic multiplier
            If player Pi has Barile on his table
                  score += enemy Barile multiplier
            If player Pi has Mirino on his table
                  score += enemy Mirino multiplier
            If player Pi has Mustang on his table
                  score += enemy Mustang multiplier
            score += max shooting distance of player Pi * enemy max
            shooting distance multiplier
            score += number of cards in player Pi's hand * enemy hand
            cards multiplier
            If player Pi has Prigione on his table
                  score += enemy Prigione multiplier
Return score
```

7.7.5 Normal moves

Parameters:

- G - All known game information

Pseudocode:

```
List M = all valid moves
List S = list with length equal to list M
H = negative infinity
For each move Mi from list M
      G2 = deep copy of G
      MakeMove(G2, Mi)
Si = score of state G2
If Si > H
      H = Si
List B = empty list
For each number Si in list S
      If Si = H
```

```
            Add Mi to list B
Return a random item from list B
```

7.7.5.1 Helper functions

EAI includes a helper function MakeMove. This function simulates making a given move and changes the game state accordingly. Since EAI does not have access to the full game information, this simulated move is slightly different from an actual move made in the game.

If the move requires drawing cards for Barile or Jourdonais's special ability, the probability of it working will be 25%. Barile or Jourdonais's special ability works if the drawn card has the hearts suit, which is true for exactly 20 cards out of the 80 Bang! cards.

We will define a new Bang! card, called "Empty", which has unknown name, suit and value. If the move includes drawing a card with unknown value, this Empty card will be used. Since all Empty cards can only go to a player's hand and EAI scoring function only uses the number of hand cards, this will not be a problem.

If the simulated move requires opponent reactions, in the simulation they will not react and therefore they will lose a life. This way, the resulting game state will have a good rating and will be likely to be chosen. This will encourage EAI to play aggressively.

Pseudocode:

MakeMove(game information G, move M)

```
Make move M in game G according to game rules
If move M requires drawing cards for Barile or Jourdonais's special
ability
      Randomly decide if Barile works or fails, with the probability
      of it working being 25%
If move M requires drawing a card from the deck
      Draw Empty card
If move M requires reactions
      If the move is Emporio
            Draw and hand out Empty cards
```

```
Else
      Every opponent makes a "do nothing" reaction
```

7.7.6 Reaction moves

The reaction picking algorithm is identical to the algorithm used in RAI and RAIR.

7.7.7 Start turn reaction moves

The start turn reaction picking algorithm is identical to the algorithm used in RAIR.

7.7.8 Training

Parameters:

- C – the original individual
- M – the number of matches per generation
- N – the number of individuals in each generation

Pseudocode:

```
While training is not stopped
      List L = list of N - 1 copies of individual C
      Mutate every individual in list L
      Add individual C to list L
      List W = list with length N, initialized to 0
      List G = list with length N, initialized to 0
      For k = 0..M – 1
            P = a random number between 4 and 7
            Create a random game with P players
            Randomly select P different individuals C1, ..., CP from
            list L
            Play out the game to the end
            For i = 0 .. P - 1
                  j = index of individual Ci in list L
                  Gj = Gj + 1
                  If individual Ci won the game
                        Wj = Wj + 1
      C = individual Cx with the highest Wx/Gx value
Return C
```

7.7.9 Advantages and disadvantages

The greatest advantage of Evolution AI is its speed, since it does not do any kind of tree searching. It also requires very little memory and computational power, and its performance does not depend on the computational power available or on the time limit for its calculations. The Evolution AI makes its decisions with more consistence than MCAI or HAI, since it does not guess any unknown game information and therefore is not affected by variable accuracy of those guesses. Also, it requires relatively little strategy knowledge, although some knowledge is necessary for setting the initial evolution settings to reasonable values. Finally, this AI is relatively simple to implement.

The main disadvantages EAI include its need for training, which is an inherent disadvantage for all evolutionary AIs. Another disadvantage is its inability to plan ahead. Finally, the Evolution AI is not able to use known information about cards in the enemy's hands.

7.8 Hybrid AI

7.8.1 Overview

Hybrid AI (HAI) exploits the fact that in order to determine how good a move is, we do not have to play the game to the end. HAI combines various elements of MCAI and EAI.

A quality of a Bang! move, or a series of moves, can be determined by its effects on health of players, the cards in their hands and the cards on their table. A series of moves is good if it increases the health of the player or his allies, add cards to their hands or tables, decreases the health of the enemies, or take away some of their cards. A series of moves is bad if it does some of the opposite.

Using the effects of a move on cards can be complicated. Obviously, some cards are more useful than others, and their usefulness depends on a great number of factors. For example, the usefulness of Bang! or Mancato! depends on the player's distance from their enemies, whether they have Barile, on their health etc. The usefulness of a Panico! card depends on the distances and on the cards of the player and everyone else. The usefulness for other cards is similarly complicated.

There is another issue with using cards to rate moves, which is overspecialization. If the AI used cards for move score, it could potentially find an excellent move for the simulated game. Unfortunately, the cards in the deck and opponent's hands might be very different in the simulation than in reality. This could lead to moves which suit the simulation but not reality.

Luckily, cards are ultimately only a tool for decreasing the health of the enemies while protecting one's own health. This means that it is possible to rate a series of moves solely on its effect on health of the players – the sequence of moves just needs to be sufficiently long. This is what HAI will use for rating moves.

7.8.2 Parallelization

HAI parallelization is identical to MCAI parallelization.

7.8.3 Individual

Like all evolution AIs, HAI uses individuals which play against each other and evolve. Each individual stores information used for picking moves.

The individual consists of:

- Tree search depth (a number between 1 and 300)
- Renegade multipliers
- Outlaw multipliers
- Sheriff multipliers
- Deputy multipliers

The multipliers for a given role consist of:

- Player's health multiplier (M_p)
- Renegade health multiplier (M_r)
- Outlaw health multiplier (M_o)
- Sheriff health multiplier (M_s)
- Deputy health multiplier (M_d)

Each multiplier is a real number from interval *<-1, 1>*.

7.8.4 Default individual

Initial individual data can be found in the electronic attachment in file "Application\Default AI individuals\Hybrid AI.txt". I chose its values based on my knowledge of the game.

7.8.5 Game state score

Score S_G of a game state G for player P is calculated as follows:

$$S_G = M_p * H_p + M_r * H_r + M_o * H_o + M_s * H_s + M_d * H_d$$

Where:

- H_p – the player's health
- H_r – the health of the Renegade
- H_s – the health of the Sheriff
- H_o – total health of all living Outlaws
- H_d – total health of all living Deputies

7.8.6 Choosing a move

A large part of the move selection algorithm is identical to MCAI algorithm. The difference between MCAI and HAI is the playout function and the backpropagation function.

Playout function pseudocode

Parameters:

- S – game situation
- C – the individual
- N – the depth of move tree where the playout begins

Returns:

- S – the game situation after N moves

Pseudocode

```
M = maximum tree search depth of the individual
While the game is not over and N <= M
      If there is at least one pending reaction request
            Select a random reaction request R
            M = a reaction move selected by RAI based on request R
      Else if there is a pending start turn reaction request
            M = a start turn reaction move selected by RAI based on
            the request
      Else
            M = a normal move selected by RAI
      Make move M and change situation S accordingly
N++
Return S
```

BackPropagate(Node N, list W)

```
Do
      N.Playouts++
      If list W contains player N.MoveMaker
            N.Wins++
      N = N.Parent
```

```
While node N has a parent
```

7.8.7 Training

Parameters:

- C – the original individual
- M – the number of matches per generation
- N – the number of individuals in each generation

Pseudocode:

```
While training is not stopped
      List L = list of N - 1 copies of individual C
      Mutate every individual in list L
      Add individual C to list L
      List W = list with length N, initialized to 0
      List G = list with length N, initialized to 0
      For k = 0..M – 1
            P = a random number between 4 and 7
            Create a random game with P players
            Randomly select P different individuals C1, .., CP from
            list L
            Play out the game to the end
            For i = 0 .. P - 1
                  j = index of individual Ci in list L
                  Gj = Gj + 1
                  If individual Ci won the game
                        Wj = Wj + 1
      C = the individual Cx with the highest Wx/Gx value
Return C
```

7.8.8 Advantages and disadvantages

Hybrid AI shares many of the advantages of MCAI. This includes its ability to take every part of game information into account, and the minimal need for strategy knowledge. In addition, HAI has several advantages over MCAI. Firstly, it needs to search much less space than MCAI, allowing it to converge to the ideal result faster. This leads to another advantage, which is better performance at low time limits for calculations or at weaker machines. Also, HAI is less affected by differences between card permutations in its simulations and in reality. Unlike MCAI, HAI is also able to

improve with experience. Finally, its limited tree search depth allows it to mimic human players better.

HAI also shares some of its disadvantages with MCAI, particularly its large needs for memory and computational power, and also its need for a given time to calculate each move. Another of its disadvantages is its need for training. In addition, this training requires a long time due to the time needed for calculating each move. Finally, its extra layers of logic and complexity make it more difficult to implement, although not significantly.

8 Comparing the AIs

8.1 Calculation

We will compare the performance of MCAI, EAI and HAI by making them play against RAIR and against each other. The number of matches will need to be sufficiently large in order to minimize the effects of the game's random elements.

Since Bang! is a team game, it would not be ideal to assign roles and AI types in each match completely at random. For example, if a stronger AI was a teammate of a weaker AI, the score of the weaker AI would be unfairly increased, and the score of the stronger AI would be unfairly lowered. This is why we will assign roles and AIs in the way described below.

In each match, we will first randomly choose the number of players, their playing order and their roles. In each Bang! game, there are three teams: the Sheriff and the Deputies, the Outlaws and the Renegade. For each team, we will randomly choose one AI type from the list of AI types available. Then we will assign the AI types to players in such a way that all players of a given team will use the team's assigned AI. For example, if team Outlaws is assigned MCAI, all Outlaws in the game will have AI type. However, a given player's AI will not know the AI types of any of its opponents. This will prevent the AIs from learning the opponent's roles based at their AI types.

In most of the tests, we will compare each AI against RAIR, which provides a reasonably good baseline. There are several reasons this AI is a good choice for testing. Firstly, it is extremely fast, allowing us to run more tests in a given time period, which in turn leads to more precise test results. Secondly, its usage of role information allows it to somewhat mimic the behavior of human players. Finally, this AI cannot improve with experience and therefore its performance will always remain constant.

We will limit the maximum time for calculating one move, since the AIs need to be able to play against human players, which means they have to make their decisions reasonably fast. In my matches against the AIs, I found time limit of 1 second to be the maximum tolerable time limit. Games with a longer time limit already felt

uncomfortably slow. This means that in the tests, we will use maximum time limits of 1 second or less.

In most tests, we will compare two AIs at a time. Since there are 3 teams in each game, two of the teams will always have the same AI type. When two teams have the same AI type, this AI's "Games participated in" value will be increased by 2.

8.2 Testing machine

All tests were be performed on the same computer with the following specifications, with no other applications running at the same time.

Testing machine specifications:

- Brand and model: HP 250 G5 Notebook PC
- Operating system: Windows 10 Home
- RAM: 8GB
- CPU: Intel Core i5-6200U, 2.30GHz (2 physical cores, 4 logical cores)
- HDD: 1TB

8.3 Test results

8.3.1 Random AI vs. RAIR

Settings:

- Total games: 5000

Results:

AI	Victories	Games participated in	Victory percentage
Random AI	2042	7500	40.84%
Random AI with roles	2958	7500	59.16%

Table 8.1

8.3.2 Monte Carlo Tree Search AI vs. RAIR

8.3.2.1 Test 1

Settings:

- Time limit for move calculations: 0.5 seconds
- Permutations tried by Monte Carlo Tree Search AI: 8
- Total games: 1000

Results:

AI	Victories	Games participated in	Victory percentage
Random AI with roles	452	1500	45.20%
Monte Carlo Tree Search AI	548	1500	54.80%

Table 8.2

8.3.2.2 Test 2

Settings:

- Time limit for move calculations: 1 second
- Permutations tried by Monte Carlo Tree Search AI: 8
- Total games: 500 (the number of games had to be lower, since each game takes a long time to play out)

Results:

AI	Victories	Games participated in	Victory percentage
Random AI with roles	221	750	44.20%
Monte Carlo Tree Search AI	279	750	55.80%

Table 8.3

8.3.3 Hybrid AI vs. RAIR

8.3.3.1 Test 1

Settings:

- Time limit for move calculations: 0.5 seconds
- Permutations tried by Hybrid AI: 16
- Hybrid AI individual: default individual
- Total games: 1000

Results:

AI	Victories	Games participated in	Victory percentage
Random AI with roles	421	1500	42.10%
Hybrid AI	579	1500	57.90%

Table 8.4

8.3.3.2 Test 2

Settings:

- Time limit for move calculations: 1 second
- Permutations tried by Hybrid AI: 16
- Hybrid AI individual: default individual
- Total games: 500 (the number of games had to be lower, since each game takes a long time to play out)

Results:

AI	Victories	Games participated in	Victory percentage
Random AI with roles	201	750	40.20%
Hybrid AI	299	750	59.80%

Table 8.5

8.3.4 Evolution AI vs. RAIR

8.3.4.1 Test 1

Settings:

- Evolution AI individual: default individual
- Total games: 5000

Results:

AI	Victories	Games participated in	Victory percentage
Random AI with roles	1623	7500	32.46%
Evolution AI	3377	7500	67.54%

Table 8.6

8.3.4.2 Test 2

Settings:

- Evolution AI individual: after 300 evolution generations (4 individuals and 200 games per generation)
- Total games: 5000

Results:

AI	Victories	Games participated in	Victory percentage
Random AI with roles	1598	7500	31.96%
Evolution AI	3402	7500	68.04%

Table 8.7

8.3.5 Monte Carlo Tree Search AI vs. Hybrid AI vs. Evolution AI

Settings:

- Time limit for move calculations: 0.5 seconds
- Permutations tried by Hybrid AI: 16
- Permutations tried by Monte Carlo Tree Search AI: 8
- Evolution AI individual: after 300 evolution generations (4 individuals and 200 games per generation)
- Hybrid AI individual: default individual
- Total games: 1000

Results:

AI	Victories	Games participated in	Victory percentage
Monte Carlo Tree Search AI	243	1000	24.30%
Hybrid AI	270	1000	27.00%
Evolution AI	487	1000	48.70%

Table 8.8

8.3.6 AIs vs. Human

AI testing would not be complete without testing their performance against a human player. I tested MCAI, EAI and HAI by playing several games against them. It should be noted that I am not an expert Bang! player. I consider my player level to be between average and experienced.

Considering the length of each game involving a human player, and the large influence of random events on the game, it would be difficult to play enough games to obtain

statistically meaningful data. Instead, I focused on the individual decisions of the AIs rather than the number of victories.

EAI and HAI seemed to play roughly at my level or slightly better, MCAI played about at my level. The AIs managed to eliminated me from the game multiple times, particularly when I started playing aggressively. All three AIs consistently avoided obviously bad moves. They could definitely be a formidable opponent for average players.

8.4 Conclusion

Evolution AI has proved to be the most successful, despite being simpler than Monte Carlo Tree Search AI and Hybrid AI. The second most successful AI is the advanced Hybrid AI, which incorporates elements of MCTS and evolution. It is followed by the slightly simpler Monte Carlo Tree Search AI, which relies on MCTS only. The next contestant is Random AI with roles, which is very simple but still uses the role guessing algorithm. Finally, Random AI is the least successful, as it is extremely simple and does not use role information at all.

The testing has shown that the tested tree searching AI algorithms may not be particularly suitable for playing Bang!. Increasing their maximum calculation time did improve their performance, but not significantly.

This low success rate of tree searching AI is likely caused by to the large number of unknown and random elements in the game. The tree searching AIs must guess the unknown information in order to do their playouts, and if the guess does not match reality closely, the AI may play very poorly.

9 Conclusion

In this work, we have analysed Bang! with regards to game theory and researched several AIs that had been successfully used in similar games. Then we have designed and implemented three different AIs, as well as the client and server application. Finally, we have compared the implemented AIs in practice. The Evolution AI proved to be the most successful, followed by Hybrid AI, Monte Carlo Tree Search AI, Random AI with roles and Random AI. The tests results have shown that tree searching algorithms are not particularly suitable for Bang! and that evolutionary algorithms are a better choice.

9.1 Possible future expansions

In the future, it would be possible to add new AIs to the implementation. These AIs could possibly be implemented in a different programming language or by other developers. The server is implemented in such a way that it impossible for a client application to cheat, either by making an illegal move or by reading secret game information, such as other player's cards. This opens up an interesting possible future use of this project, which is a programming competition. Each developer (or a developer team) would develop their own AI client. These clients would then play a sufficiently large number of matches against each other to minimize the effects of the random elements of the game, and the winner would be chosen based on the number of victories. Another option would be watching the actual matches in real time. The AIs could also play against human players.

There are many expansion packs available for Bang!, which exceed the scope of this work. These expansion packs add new game mechanics, characters and cards. Adding some of the expansion packs to the game and modifying the AIs accordingly could be an interesting addition to this work.

Another interesting expansion would be modifying the AI testing system to test how well the AIs perform in specific roles.

Bibliography

Chaslot, G. M.-B. (2010). *Monte-Carlo Tree Search.* Maastricht: Universiteit Maastricht.

Chaslot, G. M.-B., Winands, M. H., & van den Herik, H. J. (n.d.). Parallel Monte-Carlo Tree Search. 62-65.

Ciancarini, P., & Favini, G. P. (2010). Monte Carlo tree search in Kriegspiel. *Artificial Intelligence*, 672.

D. Silver, A. H. (2016). Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 1.

daVinci Editrice S.r.l. (n.d.). *Bang! F.A.Q.* Retrieved 12 10, 2016, from dV GIOCHI: http://www.dvgiochi.net/bang/bang_faq_eng.pdf

daVinci Editrice S.r.l. (n.d.). *Bang! rules.* Retrieved 12 10, 2016, from dV GIOCHI: http://www.dvgiochi.net/bang/bang_rules.pdf

Frank, I., & Basin, D. (1998). Search in games with incomplete information: a case study using Bridge card play. *Artificial Intelligence*, 87.

Guillermo, O. (2013). *Game Theory: 4th Edition.* Bingley: Emerald Group Publishing Limited.

Hansen, N., Arnold, D. V., & Auger, A. (2015). *Evolution Strategies.* Springer.

Millington, I., & Funge, J. (2016). *Artificial intelligence for games Second Edition.* Morgan Kaufmann Publishers.

List of Abbreviations

AI	Artificial Intelligence
RAI	Random AI
RAIR	Random AI with roles
MCAI	Monte Carlo Tree Search AI
EAI	Evolution AI
HAI	Hybrid AI
JSON	JavaScript Object Notation
TCP	Transmission Control Protocol
XML	Extensible Markup Language
GIMP	GNU Image Manipulation Program
UCT	Upper Confidence bound applied to Trees
MCTS	Monte Carlo Tree Search
ES	Evolution Strategies

Attachment

The attachment of the electronic version contains the following folders:

- Code
  - Folders:
    - BangClient – source code of the client application in a Visual Studio 2013 solution
    - BangServer – source code of the server application in a Visual Studio 2013 solution
- Application
  - Files:
    - Bang.exe – executable file of the client application
    - BangServer.exe – executable file of the server application
    - Evolution AI.txt – data of one Evolution AI individual after 300 generations
    - Hybrid AI.txt – data of one Hybrid AI individual
  - Folders:
    - Default AI individuals – data of initial (default) individuals for Evolution AI and Hybrid AI
- Documentation
  - Files:
    - Programmer's guide.pdf
    - Server-client communication documentation.pdf
    - Text práce.pdf
    - User's guide.pdf