Universiteit Leiden
Opleiding Informatica

AI Agents for the

Card Game Love Letter

Name: Jarno Huibers
Studentnr: s1403230

Date: July 15, 2016

1st supervisor: dr. W.A. Kosters (LIACS)
2nd supervisor: dr. J.M. de Graaf (LIACS)

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

AI Agents for the Card Game Love Letter

Jarno Huibers

Supervisors:
dr. W.A. Kosters
dr. J.M. de Graaf

Leiden Institute of Advanced Computer Science
Universiteit Leiden

Abstract

Love Letter is a card game that can be played by two to four players. The game does not use traditional playing cards but instead uses cards specifically designed for the game. Players draw cards from the same deck consisting of sixteen cards. They can use the effects of their cards to eliminate other players to win the game, or win by being the player that holds the card with the highest value when the deck is depleted.

In this thesis we look at different strategies for playing a game of Love Letter with two players, with main focus on Dynamic Programming. Using Dynamic Programming the chance of winning will be calculated for every possible configuration of a game of Love Letter. The different strategies will be compared to each other in order to try and find an optimal strategy.

It is possible to create a table containing the chances of winning and what action to perform for every possible configuration in reasonable computation time, using Dynamic Programming. The Dynamic Programming algorithm is the best strategy out of the strategies examined in this thesis.

Contents

1 Introduction

Love Letter is a card game designed by Seiji Kanai and published by Alderac Entertainment Group [1] that can be played by two to four players. Figure 1 shows the cover of the game. The cards used are not standard playing cards but are designed specifically for this game. The player's goal is to get their love letter to the princess and eliminate the other players. They can accomplish this by using the various effects of the different cards. The next section will explain the rules and the course of a traditional game of Love Letter.
The goal of this thesis was to create an agent that can play the game of Love Letter in an optimal way when the game is played with two players. In order to do this a computer program that can play Love Letter had to be created. Multiple computer players were created with different strategies such as a random player that always makes random decisions or a player that uses a Monte Carlo Tree Search algorithm to make decision. The focus of this thesis is a player that uses Dynamic Programming to find an optimal decision for every possible configuration of the game. This player will make optimal decisions assuming that the opponent also plays optimal.
The results of the different players will be compared by having them play against each other or against a player using the same strategy. As the player using Dynamic Programming is playing in an optimal way this player should give the best result in terms of win percentage when playing against the other players.

Figure 1: The cover of Love Letter, from [2].

In Section 2 the rules of the game Love Letter will be explained. We will look at the rules of a classic game of Love Letter, going over every different card and also take a look at some different versions of the game.
Next in Section 3 the different strategies for playing Love Letter will be explained. We will explain how these strategies work and how they are implemented for playing Love Letter.
In Section 4 the results of the strategies will be shown. We will show how well these strategies do when playing against each other and the results found by the Dynamic Programming algorithm.
Finally in Section 5 a conclusion will be given based on the results and we will talk about some future work that can be done to expand upon this thesis.

This thesis is a bachelor thesis written for the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University. The supervisors for this thesis are Walter Kosters and Jeannette de Graaf.

2 Game Rules

A classic game of Love Letter can be played by two to four players and uses sixteen cards in total. First the rules of classic Love Letter will be explained for up to four players even though this thesis will only focus on playing Love Letter with two players. When playing with four players all players play for themselves, so the other three players are seen as opponents for a player. The main difference between a game with two players and a game with four players is that players can choose which opponent they want to target with a card effect when playing with more than two players, but only have one choice when there is only one opponent.
Usually a player has to win a certain number of games to be declared the winner of Love Letter which might influence players to target a player who has already won the most games. This thesis will look at winning individual games of Love Letter as the player can only target one opponent.
We will also take a look at some different versions of Love Letter.

2.1 Classic Love Letter

The game Love Letter can be played by two to four players. The players share a deck consisting of sixteen cards. First the deck is shuffled and one card is put aside; this card will stay unknown to all the players so they won't know which card is excluded from the deck. Next all players will draw one card from the deck. One player will be the active player, he[1] will draw one card from the deck and then he will play a card from his hand. He always has two options, being the two cards in his hand. All players can see which cards have been played or discarded. After playing a card the next player in a given order will become the active player. There are eight different types of cards in the deck each with their own value ranging from 1 to 8. The different cards can be seen in Figure 2. A description of each card will be given, naming them C_1 through C_8 with the index number corresponding to the value of the card:

- C_1 (Guard) - 5 copies - Target one player and name a card other than C_1. If that player was holding the said card he loses the game. When a player loses the game he can no longer play any cards or be targeted by other players.

- C_2 (Priest) - 2 copies - Target one player and look at the card in his hand; only the active player can see the card.

- C_3 (Baron) - 2 copies - Target one player and compare the card in his hand to the card in the active player's hand, the player with the card

[1]The personal pronoun he refers here, and in future instances, to he or she.

with the lowest value loses the game. When it is a draw nothing happens. Only the two players involved can look at the cards when comparing.

- C_4 (Maid) - 2 copies - After playing this card no other player can select the active player as a target until the active player's next turn.
- C_5 (Prince) - 2 copies - Target one player; that player has to discard the card in his hand and draw a new card. The active player can also choose himself with this card's effect, which means he will discard the other card he was holding and draws a new card from the deck. All players can see the discarded card.
- C_6 (King) - 1 copy - Target one player; that player and the active player swap the cards in their hands.
- C_7 (Countess) - 1 copy - Playing this card does nothing, however you are forced to play this card if you have also got C_5 or C_6 in your hand.
- C_8 (Princess) - 1 copy - When a player plays or discards this card he loses the game.

The game is won by a player when he is the only player remaining. If the deck is empty and there is more than one player remaining, the player with the card with the highest value in his hand is the winner. When the values of these cards are equal the player who has discarded cards with the highest total value wins. If even these numbers are equal the game will end in a draw.

Figure 2: Eight different cards with their values, from [3].

2.2 Other Variants

There are many different versions of the game Love Letter. Most of these are just simple redesigns of the original game that use the same number of cards with the same values and effects, only with different pictures and style. However, there are some versions which use slightly different rules.
The different versions are usually based on popular movies or TV-series. In a variant based on Batman players are awarded victory tokens when they correctly guess an opponent's card with card C_1 [4]. This means that when a player has to win a certain number of games he will get closer to his goal by guessing cards with card C_1 even if he does not win that round, as normally a victory token is only given to whoever wins the round. This, however, does not change anything when looking at only a single game of Love Letter with two players.
A version based on the TV-show Archer has more differences [5]. Certain card effects allow you to choose to look at the card that was removed from the game. Also card C_7 now has a new effect where the cards of every player and the card that was removed from the game get shuffled and re-dealt to the players.
Finally a version based on the movie The Hobbit uses a deck of seventeen cards instead of sixteen [6]. This one extra card has a value of 0 and has the special ability that at the end of a round the card's value becomes 7. Also there are now two different version of card C_3, one copy of each. The first version is the same as in the traditional Love Letter game. The second version compares both cards just like the first version, only this time the player with the card with the higher value loses. The rest of the cards are identical to the traditional Love Letter game.
Another variant of Love Letter is called Big Love Letter, that can be played with up to eight players. This variant uses two sets of cards of the game Love Letter. The total number of cards and the number of copies of each different card depend on the number of players. When playing with eight players there will be thirty cards in total. There will be twice as many copies of every card except for the cards C_7 and C_8; there will always be only one copy of these cards.

3 Different Strategies

In this section we will look at some different strategies that can be used by a computer player to play Love Letter. Here we will explain how these strategies work and how they are implemented. First we will look at some simple strategies for playing Love Letter and then look at an, in some sense, optimal strategy that uses Dynamic Programming.

3.1 Simple Strategies

We will look at three different simple strategies for playing Love Letter. The first strategy is a random player who will always make random decisions. The second is a player that uses a Monte Carlo Tree Search algorithm to make his decisions and finally a more strategic player who will decide what card to play solely based on the cards he is given and whether he knows the opponent's card.

3.1.1 Random Player

The simplest strategy is a random player who will always make random decisions. First the random player will check if he is forced to play a certain card. This is the case with the effect of card C_7. Also the random player will never play card C_8 as this is the worst move possible and will always result in a loss for the active player. It is however still possible for the random player to cause himself to lose by for example playing card C_3 when the other card in his hand has a low value. When playing card C_1 the guess the random player makes will also be decided randomly. The random player will randomly choose a number between 2 and 8. If the card played is card C_1 this is the guess the random player makes. The same goes for the decision which player to target when playing card C_5; the random player will randomly choose a number, 1 or 2. Next, when the random player plays card C_5, he will target the opponent if this number was 1 and he will target himself if the number was 2.

3.1.2 Monte Carlo Tree Search

The principle of Monte Carlo Tree Search is to search for the most promising moves by playing out a certain number of random games with the current configuration. For more information on Monte Carlo Tree Search see [7]. For the implementation of the Monte Carlo player we use Basic Monte Carlo Tree Search. The player using a Monte Carlo algorithm will first check if he is forced to make a certain move. The player will never play card C_8. Also when both cards have the same value, this card will automatically be played. If the player is not forced to make a certain move he has two options, the two cards he is

holding. The program will first simulate playing the first card 1000 times. A copy is made of the current game and current configuration and the first card is played. Now the game will continue with both players playing as the random player described before. When the Monte Carlo player has no knowledge of the opponent's card the card the opponent is holding will be a random card out of the remaining cards per game. However, if the Monte Carlo Player does know which card the opponent is holding, the opponent's card will be this card for all 1000 games simulated.
The program keeps track of the number of games won by the active player, so the number of wins out of the 1000 simulated games played. The same will be done for playing the second card, 1000 random games are simulated where the active player first plays the second card and the rest of a game is played with two random players. The card with the most simulated games won out of 1000 is considered the more promising card and this is the card that will actually be played. When the number of wins for both cards is exactly the same the first card considered will be played. An example of how this algorithm works can be seen in Figure 3.

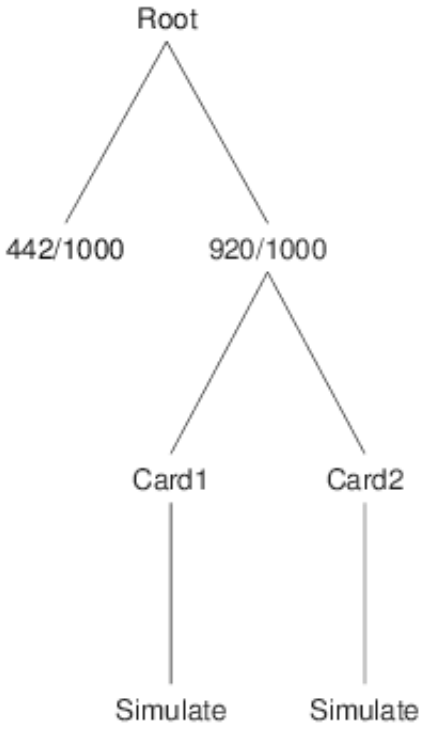

Figure 3: Monte Carlo Search Tree Example.

When the active player first has to decide on a move he is in the root node of the tree in Figure 3. Playing the first card gave him a result of 442 games won out of 1000 games played. Playing the second card gave him a result of 920 games won out of 1000 games played. This means the Monte Carlo player will play the second card and the tree continues from there. Now in his next turn the player again has to decide if he wants to play card 1 or card 2. He now has to simulate playing 1000 games for both cards again. This is indicated by the nodes Card1 and Card2 in Figure 3.
When the card played is card C_1 a guess has to be made. To decide what guess will be made 1000 games are simulated for every possible guess where the first move of the active player is playing card C_1 with the corresponding guess. The guess with the highest number of wins is the guess that will be made. If the other card the player had in his hand was not C_1, the number of wins of the value guessed with the highest number of wins will be compared to the number of wins when playing the other card. This will decide whether card C_1 will be played or the other card. When both cards the player is holding are C_1 the best guess will only be determined once.
When the card played is C_5 the player has to decide if he wants to target the opponent or himself. In a similar way as with playing C_1 1000 simulated games are played for both different options. The option with the most wins will be chosen if this number of wins is greater than the number of wins for playing the other card the player was holding that was not C_5. If both cards were C_5 the best targeting option is only determined once.

3.1.3 Strategic Player

Finally a so-called strategic player is implemented that will play according to a certain strategy based on the cards he is given and whether he knows what card the opponent is holding. First the player will check if he is forced to make a certain move and will never play card C_8. The main idea of this strategic player is to always play the card with the lowest value. The cards with lower values usually have a more immediate useful effect. Also when the deck has no more cards left, the player holding the card with the higher value wins, so the player will hold on to his higher valued cards.
If the player has knowledge of the opponent's card he will use this knowledge if it allows him to win directly after playing his next card. For example, if the player knows the opponent's card and one of the cards he is holding is C_1, he will use C_1 to guess the opponent's card and win the game. This of course only works if the opponent's card is of a value higher than 1. Another example is when the player is holding card C_3 and he knows that the value of the card the opponent is holding is lower than the value of the other card he himself is holding, he will use card C_3 to win the game. Furthermore, if the player does

not know what card the opponent is holding he will only play card C_3 if the other card he is holding is either C_6, C_7 or C_8, otherwise he will play the other card. This is the only exception where the player does not play the card with the lowest value when the player does not know the opponent's card.
The guess that is made when playing card C_1 is decided by looking at the number of cards that are left of each copy. The player will always guess the card with the most copies left if he does not know what card the opponent is holding. If multiple cards have the same number of copies left the player will guess the card with the lowest value.
If the player plays card C_5 he will always target the opponent. The only time the player will target himself is when the opponent played card C_4 in his previous turn, the player now has no other choice than to target himself.

3.2 Dynamic Programming

Dynamic Programming is used to solve a large problem by reducing it to smaller, similar problems that are easy to solve, and this is done in such a way that the smaller problems are solved first and only once. For more information on Dynamic Programming see [8, Chapter 15].

3.2.1 Love Letter Implementation

When playing Love Letter, if a player knows his move will be the last turn it can be decided what card will be the best choice and how big his chances of winning are. A function is used that will return 1 when the chances of winning are 100%, 0.5 when the chances of winning are 50% and 0 when the chances of winning are 0%. This function is used for playing Love Letter with two players. When the game will result in a draw 0.5 is given as the return value. The function does not look at the total value of discarded cards and will result in a draw when the last cards of both players are the same. When the current turn is not the last turn the function will still be called but will be called recursively until a return value is given.
The function definition is as follows: $f(b_1, b_2, c, A, maid, known, p)$. The parameters b_1 and b_2 are integers in {1,2,3,4,5,6,7,8} that indicate the values of the two cards the player calling this function has in his hand; these are his two options. The parameter A indicates the number of remaining cards and the values of these cards. The remaining cards consist of the cards that are still left in the deck, the card in the opponent's hand and the card that has been put aside in the beginning of the game. The parameter A actually is a vector that consists of 8 different parameters which all indicate the number of cards of a certain value, so $A = (a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8)$. If there are still 3 copies left of card C_1, then a_1 will be 3. The parameter c indicates the card that the other player is holding in his hand; the player calling the function does

not know the value of this card, but this value will be used to calculate the correct return value. This also means that card c is included as a remaining card indicated by A. The value of c can also be 0, this is the case when the opponent has not yet made a move. Now the value of c will not be necessary to calculate the correct return value. The parameter *maid* is a Boolean that indicates if the other player has played card C_4 in his previous turn, as this will affect the course of the game. The parameter *known* is a Boolean that indicates whether the active player has knowledge of the card in the opponent's hand. This means that when *known* is set to true, the active player knows the value of card c. When the value of c is 0 the Boolean *known* will never be set to true. Finally parameter p is an integer in {0,1,2} that indicates whether the opponent has knowledge of the cards of the active player. When the value of p is 1 the opponent has knowledge of card b_1, when the value of p is 2 the opponent has knowledge of card b_2 and when the value of p is 0 the opponent has no knowledge of the active player's cards.

When playing card b_1 a certain gain value is calculated and the same is done for playing card b_2. This gain value for card b_1 is calculated by a certain formula. This formula calculates the gain value of playing card b_1 by looking at all the possible cards the opponent could be holding and how big the chances of winning are for the player playing card b_1. This chance of winning will be multiplied by the chance that the opponent was actually holding that card and because this will be done for every possible card the summation of these values becomes the gain value.
To calculate the chance of winning a function r is used that can be defined as follows: $r(b_1, b_2, i, A, maid, guess, known, p, prince)$. The parameters b_1 and b_2 indicate the values of the cards the active player is holding and are the same as in the call to f. Function r simulates what happens when the player chooses to play card b_1, so the first parameter of r is the card that is being played. When the gain value for playing card b_1 is calculated the value of b_1 in r is the same as the value of b_1 in f, however if the the gain value for playing card b_2 is calculated the value of b_1 in r is the same as the value of b_2 in f. The parameter i indicates the card the opponent is currently holding. Parameters A, *maid*, *known* and p are the same as in the call to f. Parameter *guess* is an integer in {2,3,4,5,6,7,8} used to indicate the guess that is made when the active player plays card C_1. When this guess is the same card as the one the opponent is holding the player playing C_1 wins. This value is only used if the card played is actually card C_1, so if the value of b_1 is 1. All different possible guesses, so all cards other than C_1 have to be taken into consideration when calculating the chance of winning. This means that r has to be used multiple times when the card played is C_1. Parameter *prince* is an integer that indicates whether the active player targets the opponent or himself when playing card

C_5. The value of *prince* is 1 when the active player targets the opponent and 2 when the active player targets himself. Note that when the opponent played card C_4 in his previous turn he cannot be targeted and the player playing card C_5 will have to target himself.
When the result of function r is not a direct win or direct loss for the active player the function f has to be called recursively, it is now called for the other player. Because the other player does not know what card the first player is still holding he will look at every possibility, however when calculating the gain value for the first player the card he was actually holding needs to be taken into consideration. This is done when the value of c in the recursive call to function f is not 0 but *known* is false. The function r will now be used again to calculate the true gain value, which is explained later.

3.2.2 Operation Example

An example of the use of the formula is as follows. Suppose there are five cards left and these cards are C_2, C_3, C_6, C_7 and C_8, one copy of each. Player 1 has two cards in his hand, these are C_3 and C_6, and has to decide which card to play. The remaining cards are C_2, C_7 and C_8. One of these cards is the last card in the deck, one of these cards is the card the opponent is holding and the other card is the card that was put aside at the beginning of the game. Suppose that in the previous turn the maid, card C_4, was not played and neither player has knowledge of the other player's cards, the call to function f will be as follows:

$$f(3, 6, 0, 0, 1, 0, 0, 0, 0, 1, 1, false, false, 0).$$

The card that the opponent possibly has in his hand will be called i and the total number of possible remaining cards will be indicated by n. In this case n will be equal to 3 because there are 3 cards remaining, one copy of C_2, one copy of C_7 and one copy of C_8. Also p_i/n will be used to indicate how many of the remaining cards have value i, so p_i indicates the number of cards with value i. The function will first look at the gain value from playing card C_3. The value of p_i/n will be 0 for $i = 1, i = 3, i = 4, i = 5$ and $i = 6$. When $i = 2$ the value of p_i/n will be $1/3$ and the function

$$r(3, 6, 2, A, false, 2, false, 0, 1)$$

will return a value of 1 because of a direct win. Note that when the parameters *guess* and *prince* of function r are not used they are given as their lowest possible value. When $i = 7$ the value of p_i/n will be $1/3$ and the function

$$r(3, 6, 7, A, false, 2, false, 0, 1)$$

will return a value of 0 because of a direct loss. The same result is given when $i = 8$, meaning that the total gain value for playing C_3 is $1/3$.

The gain value for playing C_6 will be a bit more difficult to determine. We first look at the value gained when $i = 2$. The value of p_i/n will be $\frac{1}{3}$. Next the call

$$r(6, 3, 2, A, false, 2, false, 0, 1)$$

will not result in a direct win or direct loss, so the function f will have to be called again. The function r did however swap the values of the cards the players are holding because of the effect of card C_6, so player 1 is now holding card C_2 and player 2 is now holding card C_3. Now the opponent will make a move. It is important to look at the card the opponent will draw in his turn to determine what his two options are. The card the opponent possibly draws in his next turn will be called j. First we look at $j = 7$. The call to function f now becomes,

$$f(3, 7, 2, 0, 1, 0, 0, 0, 0, 0, 1, false, true, 1).$$

This function call will give a value of 1, because the second player will always win when the first player is holding card C_2. To get the chances of winning for the first player this value needs to be subtracted from 1 so in this case the chance of winning for the first player will be $1 - 1$, or 0. Next look at $j = 8$; the call to f now becomes,

$$f(3, 8, 2, 0, 1, 0, 0, 0, 0, 1, 0, false, true, 1),$$

which has a predetermined return value of 1, meaning that the chances of winning are 0 for the first player. The total gain value when $i = 2$ becomes, $1/3 * (1/2 * 0 + 1/2 * 0)$, or 0.
Next look at $i = 7$ and $j = 2$; the call to f becomes

$$f(2, 3, 7, 0, 0, 0, 0, 0, 0, 1, 1, false, true, 1),$$

giving a value of 0 because playing either C_2 or C_3 will always result in a loss. When playing card C_3, the player will lose because the opponent's card has a higher value than the other card in the player's hand, C_2. When playing card C_2 the player will lose because the deck will be depleted and the card the opponent is holding will have a higher value than the card the player is holding, C_3. When $j = 8$ the call to f becomes

$$f(3, 8, 7, 0, 1, 0, 0, 0, 0, 0, 1, false, true, 1),$$

giving a value of 1 because playing card C_3 will result in a win. The total gain value when $i = 7$ becomes, $1/3 * (1/2 * 1 + 1/2 * 0)$, or $1/6$.
Finally look at $i = 8$ and $j = 2$; the call to f becomes

$$f(2, 3, 8, 0, 0, 0, 0, 0, 0, 1, 1, false, true, 1),$$

giving a value of 0 because playing either C_2 or C_3 will always result in a loss. When $j = 7$ the call to f becomes

$$f(3, 7, 8, 0, 1, 0, 0, 0, 0, 0, 1, \mathit{false}, \mathit{true}, 1),$$

giving a value of 0, because the second player will always lose if the first player is holding card C_8. The total gain value when $i = 8$ becomes $1/3 * (1/2 * 1 + 1/2 * 1)$ or, $1/3$. This means the total gain value for playing card C_6 becomes $0 + 1/6 + 1/3 = 1/2$.
In conclusion, the total gain value for playing card C_3 was $1/3$ and because $1/3 < 1/2$ the player will choose to play card C_6 instead of C_3.

3.2.3 The Used Formula

The general formula for calculating the gain value when playing b_1 is described as follows :

$$\mathit{gain}(b_1) = \sum_{i=1}^{8} \frac{p_i}{n} r(b_1, b_2, i, A, \mathit{maid}, \mathit{guess}, \mathit{known}, p, \mathit{prince})$$

with $r(b_1, b_2, i, A, \mathit{maid}, \mathit{guess}, \mathit{known}, p, \mathit{prince}) = 1$ in case of a direct win for the active player, $r(b_1, b_2, i, A, \mathit{maid}, \mathit{guess}, \mathit{known}, p, \mathit{prince}) = 0$ in case of a direct loss and

$$\begin{aligned} &r(b_1, b_2, i, A, \mathit{maid}, \mathit{guess}, \mathit{known}, p, \mathit{prince}) \\ &\quad = \sum_{j=1}^{8} \frac{q_j}{n-1} (1 - f(i, j, b_2, A - i - j + b_2, \mathit{maid}, \mathit{known}, p)) \end{aligned}$$

for other instances.

The parameters used in the recursive call to f will be explained later. The formula looks at the chance that the opponent has a certain card i, indicated by p_i/n, and multiplies this by the return value of the function r. Eventually the formula gives a value between 0 and 1 which indicates the chance of winning when playing card b_1. The same will be done for playing card b_2 using the same formula only now with every instance of b_1 replaced by b_2 and vice versa. The result will now be saved as $\mathit{gain}(b_2)$ instead of $\mathit{gain}(b_1)$. The first parameter of function r is the card that is played but in function f both cards can still be played. The result of the function f will be the following:

$$f(b_1, b_2, c, A, \mathit{maid}, \mathit{known}, p) = \max(\mathit{gain}(b_1), \mathit{gain}(b_2))$$

The value of b_1 will always be lower than or equal to the value of b_2. The order

in which b_1 and b_2 appear in f has no influence on the outcome of function f, so in order to avoid calculating function f multiple times for the same state of the game the first parameter will always be lower than or equal to the second parameter. When the value of c in the call to function f is 0, meaning that the active player has no knowledge of his opponent's card, the result of the summation of using function r will be returned as stated in the formula, however if the value of c was not 0, the function r will be used again, only this time no summation and fraction is used before calling r; the value of i will be the value of c and the card played will be b_1 if b_1 gave a gain value higher than or equal to b_2, otherwise the card played will be b_2. The result of this function r will then be returned without the use of summations or fractions. There is an exception when the gain values of b_1 and b_2 are the same but the values of using r for the second time for b_1 and b_2 are different. In this case the original value of the formula will be returned.

The variable i in the formula indicates the value of the card that the opponent is holding; the formula will look at every possible card. The variable j indicates the card the opponent will draw during his next turn. Furthermore p_i stands for the number of copies of card i that are still remaining, so they are either left in the deck, in the opponent's hand or were put aside at the beginning of the game; q_j stands for the number of copies of j that are still remaining with one copy of card i no longer a remaining card. The variable n stands for the total number of remaining cards at the start of the active player's turn, so n is the summation of the elements of vector A;

$$n = a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8$$

The function r is used to simulate playing card b_1. The parameter b_2 is the other card the player is holding in his hand and the parameter i is the card the opponent is holding. When the function r results in a direct win for the player the gain value 1 will be returned for that value of i. When the function r results in a direct loss for the player the gain value 0 will be returned for that value of i. When neither of these cases occur the formula will look at the possibilities for the other player and the function f is called recursively. Function r will handle the effects that the different cards that are played can have. This means that the values of *maid*, *known* and p can change based on what happens in function r.
The function f will be called when the second player is holding card i and draws card j. The parameter c will get the value of card b_2. The parameters a_i and a_j corresponding to cards i and j will be decreased by 1 because these cards are not remaining cards to the second player and the parameter a_{b_2} corresponding to card b_2 will be increased by 1 because this card is unknown to

the second player and is seen as one of the remaining cards. This is indicated by $A - i - j + b_2$ in the formula. The Boolean *maid* will be set to true if b_1 was card C_4 and will otherwise be set to false. Boolean *known* will be true if the knowledge of the first player's card was gained by the second player or if the second player already knew the first player's card and the opponent did not play that particular card. Integer p will indicate whether the first player gained or already had knowledge of the second player's card. The gain value that will be returned will be 1 minus the value of f. This is because function f will return the chances of winning for the second player so 1 minus the value of f will be the chances of losing for the second player, or the chances of winning for the first player.
This call to the function f will give a value based on the parameters of the function call. This is a value that has already been calculated by an earlier call to function f according to the principle of Dynamic Programming. First the values of function f will be calculated with the smallest number of remaining cards. When the current turn is the final turn of a game the chances of winning can be calculated. These values are stored in a big table. When f has to be calculated with a bigger number of remaining cards the function will look in the table when f gets called recursively as the value will already be there, because there is a smaller number of remaining cards in the recursive call to f. This means that the function will not have to calculate that value again. The new value of f will then also be saved in the table. This means that bottom up Dynamic Programming is used where a recursive call to f means that the program will look in the table for an already calculated value of a smaller problem.

The card for which the formula gives the highest gain value will be the card that is actually played. This formula is used during regular instances but there are certain exceptions based on the value of the card played b_1 or b_2. The function r will handle these exceptions.

3.2.4 Handling Different Card Effects

We now distinguish the eight cases for card b_1:

- When the card played is C_1 the formula will be used 7 times, once for each possible guess that can be made when playing card C_1. This means that seven different gain values will be given. Every time the formula is used a different value *guess* will be used, corresponding to the possible guesses. Now when the value of *guess* is equal to the value of i in the formula, the function r will indicate a direct win and 1 will be returned. If *guess* is not equal to i the function f will be called recursively as stated

in the formula. Say that card b_1 is C_1 and card b_2 is a different card, there will now be eight different options instead of the usual two.

- When the card played is C_2 the active player will gain knowledge of the card that his opponent is holding. This means that when the opponent does not play this card in his next turn the player will not have to look at every possible card the opponent could be holding because he already knows what card the opponent is holding. The Boolean *known* indicates whether the active player knows his opponent's cards and integer p is used to indicate whether the opponent knows one of the active player's cards. Now the gain values for playing cards b_1 and b_2 have to be calculated for only that instance of variable i.

- When the card played is C_3 the card that is not played will be compared to card i in function r, so if b_1 is played b_2 will be compared and vice versa. If b_1 is played and the value of b_2 is higher than the value of i the return value of r will be 1. If b_2 is smaller than i the return value of r will be 0. When the value of b_2 is equal to the value of i the formula will be used as normal for that part of the calculation. Furthermore when this is the case the players have knowledge of each other's cards, meaning that as long as the opponent does not play a card with the same value of the card a player has knowledge of, the same applies for calculating the gain value as with playing card C_2.

- When the card played is C_4 the Boolean *maid* in the function f will be set to true, otherwise this value will always be false. When this Boolean is true the call to function r will never result in a direct win or a direct loss in that call to function f, no matter what card is played. For instance, the exceptions previously described when playing card C_1 or C_3 will not take effect and the formula will be used as normal. When *maid* is set to true the active player still has to play and discard a card even though the card will not have its usual effect when targeting the opponent. The only exception is card C_5 where the active player will be forced to choose himself as a target. After the second player has played a card the Boolean *maid* will become false, unless the card played was also C_4.

- When the card played is C_5 the formula will be used two times because the player can target his opponent or himself. This is similar as with playing card C_1, but now two different gain values will be given. They are calculated by using the variable *prince* to indicate whether the player targets himself or the opponent. When *prince* has value 1 the formula will change slightly because the card the opponent is holding will be discarded and a new card is drawn. This means that the formula has to look at the card the opponent discards, the new card he then draws and

the card he will draw in his next turn. The formula when playing card b_1 now becomes:

$$gain(b_1) = \sum_{i=1}^{8} \frac{p_i}{n} r(b_1, b_2, i, A, maid, guess, known, p, 1)$$

with $r(b_1, b_2, i, A, maid, guess, known, p, 1) = 1$ in case of a direct win for the active player, $r(b_1, b_2, i, A, maid, guess, known, p, 1) = 0$ in case of a direct loss and for other instances

$$r(b_1, b_2, i, A, maid, guess, known, p, 1) =$$

$$\sum_{k=1}^{8} \frac{m_k}{n-1} \sum_{j=1}^{8} \frac{q_j}{n-2} (1 - f(k, j, b_2, A - i - j - k + b_2, maid, known, p))$$

.

When *prince* has value 2 the formula will also change because the other card the player was holding will be a new card, so the formula has to look at the possibilities of what this card could be. The formula when playing card b_1 now becomes:

$$gain(b_1) = \sum_{i=1}^{8} \frac{p_i}{n} r(b_1, b_2, i, A, maid, guess, known, p, 2)$$

with $r(b_1, b_2, i, A, maid, guess, known, p, 2) = 1$ in case of a direct win for the active player, $r(b_1, b_2, i, A, maid, guess, known, p, 2) = 0$ in case of a direct loss and

$$r(b_1, b_2, i, A, maid, guess, known, p, 2) =$$

$$\sum_{k=1}^{8} \frac{m_k}{n-1} \sum_{j=1}^{8} \frac{q_j}{n-2} (1 - f(i, j, k, A - i - j, maid, known, p))$$

for other instances.

The variable m_k represents the number of copies of card k left remaining with card i respectively card b_2 no longer a remaining card. The value of the variable q_j will now be the number of remaining copies of card j with one copy of cards i and k no longer remaining. When *prince* is 1 the function r will result in a direct win if the value of i is 8, otherwise the function f will be called recursively as described above. When *prince* is 2 the function r will result in a direct loss if the value of b_2 is 8, otherwise the function f will be called recursively as described above.

- When the card played is C_6 the function r will swap the value of the other card the active player is holding with the card the opponent is holding before calling function f. These are the cards b_2 and i when the card played was b_1. Furthermore the players now have knowledge of each other's cards meaning that the same applies for calculating the gain value as with playing card C_2.

- When the card played is C_7 the formula will be used as normal, however when card b_1 is C_7 and card b_2 is either C_5 or C_6 card b_1 will have to be played. When card b_2 is C_7 and card b_1 is C_5 or C_6, card b_2 will have to be played. Only the gain value for playing C_7 will have to be calculated and no gain value will be calculated for playing C_5 or C_6.

- When the card played is C_8 the gain value will always be 0 and no calculations will be needed.

4 Results

In this section we will take a look at the table that is built using the Dynamic Programming algorithm. We will also look at the results the different players get when playing against each other, by having them play a large number of games. This way it will be possible to see which algorithm gets the best results and also to check whether the beginning player has an advantage over the player going second. First we will look at the results of the more simple strategies and then at the results of the Dynamic Programming algorithm.

4.1 Simple Strategies

First the random player will play one million games against another random player. The program keeps track of the total number of games won by the beginning player and the number of games won by the second player. The games that are neither won by the first player nor the second player resulted in a draw. Each game uses a random permutation of the deck and starting cards. The results can be seen in Table 1.

Starting Player	Second Player
516 016	483 867

Table 1: Number of games won of random versus random.

When both players use a random strategy the player who makes the first move has a slight advantage over the player making the second move. The percentage of games won by the first player however is not much larger than the percentage of games won by the second player; 52% won by the first player against 48% won by the second player.
Next we look at a player using the Monte Carlo algorithm. This player will also play one million games against a random player. The results can be seen in Table 2. The computation time of playing one million games was $181m43.821s$.

Monte Carlo Player	Random Player
526 314	473 595

Table 2: Number of games won of Monte Carlo versus random.

In all of the games played the player using the Monte Carlo algorithm was the player to make the first move. The results show that the Monte Carlo player does slightly better than the random player when playing against another random player. Because the simulated games the Monte Carlo player plays are played randomly, the Monte Carlo player does not do that much better than the random player. However if a Monte Carlo player can make a move that will

win him the game immediately he will always make that move.
Finally we look at the strategic player, this player will also play one million games against a random player. The results can be seen in Table 3.

Strategic Player	Random Player
717 719	282 243

Table 3: Number of games won of strategic versus random.

The strategic player was the first player in all one million games played. The strategic player does a lot better than the random player or the Monte Carlo player when playing against a random player. The strategic player wins just over 70% of the games played. When playing Love Letter luck can play a big factor on the outcome of a game. This is why the random player still wins almost 30% of the games played.

4.2 Dynamic Programming

Using the described formula it is possible to create a large table that contains the chances of winning for the active player for all possible configurations of the game. The computation time for creating this table is $13.500s$, so the table can be created in very reasonable computation time. The total number of non-zero entries the table contains is 2 604 215. These entries are all configuration of the game Love Letter that can possibly be reached and entries that are used to calculate the chance of winning for other configurations. This table is contained in a multidimensional array that can hold around 24 million values. An array element has the following structure:

$$\texttt{Chance}[b_1]\,[b_2]\,[c]\,[a_1]\,[a_2]\,[a_3]\,[a_4]\,[a_5]\,[a_6]\,[a_7]\,[a_8]\,[\mathit{maid}]\,[\mathit{known}]\,[p]$$

and will represent $f(b_1, b_2, c, A, \mathit{maid}, \mathit{known}, p)$. This means that an entry of the array contains the chance of winning when the active player has cards b_1 and b_2 and the opponent has card c; the value c is 0 if the active player does not know the opponent's card. The remaining cards correspond to a_1 through a_8, *maid* is 1 if the opponent played card C_4 in the previous turn, *known* is 1 if the active player knows what card the opponent is holding, c will not be zero if this is the case and finally p will be 1 if the opponent knows the active player's card b_1, 2 if the opponent knows the active player's card b_2 and 0 if the opponent has no knowledge of the active player's cards. Only possible configurations need to be calculated.
The card that has to be played in a certain situation is also saved in a similar array: this is the card that gives the highest chance of winning. When the chances of winning of both cards are equal, the first card, so the card with the lowest value, will be chosen. The same is done for what guess to make when

the card chosen to be played is C_1 and who to target when the card chosen to be played is C_5. When multiple guesses give the same chance of winning the player will guess the card with the lowest value and when both options when playing C_5 give the same result the player will target the opponent.
Table 4 shows the results when the player has to make the very first turn of a normal two player game of Love Letter.

The first column indicates the number of cards that are still remaining. In the beginning of the game the number of cards remaining will be fourteen. These include the twelve cards still left in the deck, the card the opponent is holding and the card that has been taken out of the deck at the beginning of the game. These cards all have to be considered when making a decision.
The second and third column indicate the cards the active player is holding b_1 and b_2. The lowest card will always be indicated by b_1 to avoid getting multiple entries in the table that are the same configuration as the order of the cards the player is holding does not matter for the resulting chance of winning.
The fourth column shows the result of function f, this is the chance of winning rounded to 6 decimal places. This number indicates the highest probability of winning out of the two possible cards that can be played.
The fifth column shows which of the two cards a player using this table will play. This is the card that gave the highest probability of winning out of the two cards b_1 and b_2. The probability indicated in the fourth column corresponds to playing the card indicated in the fifth column.
The sixth column shows the guess that has to be made when the card that has to be played, so the card from the fifth column, is C_1. The probability from the fourth column corresponds to this guess. The seventh column indicates what player to target when the card played is card C_5. A 1 in this column indicates the player has to target his opponent and a 2 indicates the player has to target himself with the effect of card C_5.
Most probability values are around 0.5, however there are some clear exceptions. When the player has card C_3 and card C_8 in his hand and C_4 was not played by his opponent, his chances of winning will be 100%. This is because there is only one card C_8 so if you compare this card to another card, C_8 will always be the highest card. When the active player starts with two copies of card C_3 his chances of winning are exactly 50% because half of the remaining cards have a value lower than 3 and the other half have a value higher than 3.

n	b_1	b_2	*Chance*	Play	Guess	Target
14	1	1	0.523280	1	3	
14	1	2	0.474567	2		
14	1	3	0.395335	1	2	
14	1	4	0.510439	1	3	
14	1	5	0.565033	1	2	
14	1	6	0.635893	1	2	
14	1	7	0.683812	1	2	
14	1	8	0.709889	1	2	
14	2	2	0.487082	2		
14	2	3	0.413797	2		
14	2	4	0.525749	4		
14	2	5	0.464601	2		
14	2	6	0.626971	2		
14	2	7	0.667182	2		
14	2	8	0.638904	2		
14	3	3	0.500000	3		
14	3	4	0.596214	3		
14	3	5	0.736906	3		
14	3	6	0.857143	3		
14	3	7	0.928571	3		
14	3	8	1.000000	3		
14	4	4	0.547819	4		
14	4	5	0.487741	4		
14	4	6	0.599004	4		
14	4	7	0.608690	4		
14	4	8	0.651215	4		
14	5	5	0.565066	5		1
14	5	6	0.552904	5		1
14	5	7	0.415273	7		
14	5	8	0.626603	5		1
14	6	7	0.618673	7		
14	6	8	0.542204	6		
14	7	8	0.624939	7		

Table 4: Chances of winning at the start of the game.

In order to show what the results will look like we will use the example from Section 3.2.2, where the player was holding cards C_3 and C_6 and there were three cards left, C_2, C_7 and C_8. The results will give:

```
[3] Chance[3][6][0][0][1][0][0][0][0][1][1][0][0][0]: 0.5 Play 6
```

The first number 3 on the left side indicates the number of cards that are remaining. This is the value of n in the table. The other numbers between the square brackets indicate the configuration of the array described before. The player is holding cards C_3 and C_6 and does not know the value of the card the opponent is holding. The card with the lowest value is always seen as the first card and the card with the higher or equal value is seen as the second card, as stated before. The last three numbers are all 0 because in the example we assumed that both players had no knowledge of each other's cards and the card C_4 was not played in the previous turn.
The first number next to the colon is the chance of winning for the active player for that particular configuration assuming that the opponent plays in an optimal way similar to the player. This means that the chance of winning can be higher if an opponent is not playing in an optimal way. This number will always be between 0 and 1, as this is the probability of winning. The best chance of winning for this particular configuration is 0.5, which corresponds to the value found in Section 3.2.2.
The card with the highest probability of winning is the card that has to be played. This card is indicated by "Play" followed by the card that has to be played. When this card is C_1 the card that has to be guessed is indicated by "Guess" and when the card that has to be played is C_5 the value after "Target" indicates whether the opponent or the player himself should be targeted. This number is 1 if the opponent has to be targeted and 2 if the player himself should be targeted. In this case card C_6 has to be played as shown in Section 3.2.2 and no guess has to be made, nor does the player need to decide who to target.

In order to see how well a player would do if he would always plays the card indicated by the table in the corresponding situation a player using the table played one million games against a random player. The results of playing one million games versus a random player are seen in Table 5.

Player using table	Random Player
723 929	276 026

Table 5: Number of games won of Dynamic Programming versus random.

The player using the table was the starting player. The player using the table wins more than 70% of the games played. These results are similar to the results

from the strategic player versus a random player. When you look at the cards that need to be played in the fifth column of Table 4 almost always the card with the lowest value gives the highest probability of winning, which was part of the strategy used by the strategic player. The percentage of games won by using Dynamic Programming is still higher than that of the strategic player, however.
When the player has to play against another player that uses the table, these are the results:

Starting Player	Second Player
569 022	430 949

Table 6: Number of games won of Dynamic Programming versus Dynamic Programming.

These results show that the beginning player has a bigger chance of winning than the second player when both players play in an optimal way according to the table. When looking at the results in Table 4, most beginning situations have a probability higher than 0.5 meaning that the game will result in a win for the first player. This is reflected in the result of playing one million games between two players using the table as the first player wins around 57% of the games. When you compare the percentage of games won by the starting player to that of the starting player when both players are playing randomly, you can see that the influence of making the first move is greater on the number of games won when the players are playing using the table, so playing in an optimal way.

Because the percentage of games won against a random player by the strategic player and the player using Dynamic Programming are similar we now look at the number of games won when the player using Dynamic Programming will play directly against the strategic player. The results can be found in Table 7.

Player using Table	Strategic Player
619 072	382 081

Table 7: Number of games won of Dynamic Programming versus strategic.

The starting player is the player using Dynamic Programming. Even though both strategies got similar results when playing against a random player the player using Dynamic Programming does quite better than the strategic player when they play directly against each other. However the beginning player has an advantage over the second player as shown before, so we will also look at what happens when the strategic player is the starting player. Table 8 will show the results when the strategic player was the starting player.

Strategic Player	Player using Table
509 168	490 808

Table 8: Number of games won of strategic versus Dynamic Programming.

The number of games won by both players are now a lot closer than when the player using Dynamic Programming was the beginning player. The strategic player will now win slightly more games than the player using Dynamic Programming, however this percentage of games won is much lower than that of the player using Dynamic Programming when he was the starting player. This percentage is also lower than the percentage of games won by the player using Dynamic Programming when he played against another player using Dynamic Programming and was the starting player.

In order to test the validity of the values of the array the player will play one million games against another player that uses the table, where the two cards the first player has in the beginning are set. The chance of winning indicated in the table shows the chance a player will win when he starts with two specific cards. So if you play one million games, where the first player always starts with the same exact two cards, the percentage of games he wins should be close to the chance of winning indicated in the table. For example if the player always starts with one copy of card C_1 and one copy of card C_8 the results of playing one million games are shown in Table 9.

Starting Player	Second Player
723 094	276 906

Table 9: Player starts with 1 and 8.

The percentage of games won by the starting player is around 72%, which is only slightly higher than the probability of winning seen in Table 4, which is around 70%. There are no games that result in a draw, this is because a game can only result in a draw when card C_8 is the card that was taken out of the deck. Because the first player is holding card C_8, this is not the case and the game cannot result in a draw. When the cards the first player starts with are C_1 and C_7 the results of playing one million games can be seen in Table 10.

Starting Player	Second Player
690 221	309 713

Table 10: Player starts with 1 and 7.

The percentage of games won by the starting player is now lower than when the starting player had C_1 and C_8, just like indicated in Table 4. The starting player wins around 69% of the games and the table indicates a winning chance of around 68%.

4.3 Full Comparison

Table 11 shows the results of every different strategy playing against players using every other strategy and a player using the same strategy. The strategy on the left side of the table is the player that made the first move. The table shows the number of wins by the starting player out of one million games played.

X	Random	Monte Carlo	Strategic	Dynamic Programming
Random	516 016	517 986	391 792	345 708
Monte Carlo	526 314	531 099	361 722	363 481
Strategic	717 719	709 305	545 783	509 168
Dynamic Programming	723 929	656 660	619 072	569 022

Table 11: Full comparison results.

The table clearly shows that the starting player always has an advantage. When a player is playing against another player using the same strategy the percentage of games won by the starting player is always over 50%. The better the strategy the bigger this advantage becomes, as the percentage of games won by the starting player gets higher when the strategy gets better.
The player using Dynamic Programming gives the best results as this player has the most wins when playing against almost every other player. Only the strategic player does better than the player using Dynamic Programming when playing against a Monte Carlo player. The player using Dynamic Programming expects optimal play from his opponent whereas the strategic player does not. Luck is also a factor that determines the outcome of a game, hence why a random player can still win a fair number of games against clearly superior players. When playing directly against each other the player using Dynamic Programming does get quite better results than the strategic player. Both players win more games when they are the starting player, but the player using Dynamic Programming wins significantly more games when he is the starting player than the strategic player does, when he is the starting player.

5 Conclusion and Future Work

In this thesis we have looked at different strategies for playing the card game Love Letter. First we looked at some simpler strategies, a random player, a player that uses a Monte Carlo algorithm and a player that plays strategically by following certain guidelines to make his decisions based on the cards he is holding. The main focus was on a player who uses Dynamic Programming to determine the chance of winning for every possible configuration of the game and what card needs to be played to get the highest probability of winning.
The results have shown that out of the strategies that are used in this thesis, the strategy of using Dynamic Programming to determine the move with the highest probability of winning gives the best results, in terms of games won. Because this strategy always makes the decision that gives the player using this strategy the highest probability of winning, this strategy can be considered an optimal strategy.
The results also show that it is possible to calculate the chance of winning for every possible configuration of Love Letter with two players in reasonable computation time.
Love Letter is a game that is partly influenced by luck. Both the strategic player and the player using Dynamic Programming had similar results when playing against a random player, however when they played against each other directly the player using Dynamic Programming did significantly better. Because of the luck involved a random player will always win some games, even when playing against an opponent that plays optimally.

In this thesis we have looked at playing Love Letter with two players. Because the game can be played with up to four players, for future work one could first increase the number of players to three and later even to four. With three players the number of configurations already makes a huge increase. If one were to try using a Dynamic Programming algorithm, the algorithm not only has to check a lot more possibilities when calculating the chance of winning but also has more options, because the player has to decide who he wants to target for every card.
Another idea for future work is to look at information a player can acquire without the direct effect of a card. A good example of this is when an opponent plays card C_7, there is a good chance he also had either card C_5, C_6 or C_8 in his hand, as these cards force a player to play C_7. Playing card C_7 otherwise does not have many advantages. When the opponent plays a certain card the player can wonder why he would play that card instead of the other card in his hand and this way determine what the other card could be.
Finally one could look at the different version of Love Letter described in Section 2.2. Big Love Letter is a version that can be played with up

to eight people where the number of copies of each card differs depending on how many players there are. Perhaps more interesting is the version based on The Hobbit, as this version is still similar to classic Love Letter with only a few extra cards. Especially interesting is the new card of value 3 that compares cards with the lowest valued card winning as this will make the higher valued cards less powerful and at the same time gives more use to lower valued cards. This might make a strategy of always playing the lowest valued card less efficient and drawing a high valued card might not be as advantageous. Looking at the different versions can be done in the same way as we have looked at classic Love Letter. You can create a table in the same way using Dynamic Programming without much extra computation time, only the function that handles the card effects changes.

References

[1] Alderac Entertainment Group, Love Letter, https://www.alderac.com/loveletter [retrieved 29.06.2016].

[2] BoardGameGeek, Love Letter, https://boardgamegeek.com/boardgame/129622/love-letter [retrieved 29.06.2016].

[3] S. Coggins, Love Letter Review, BoardGameGeek https://boardgamegeek.com/thread/1007073/radio-review-31-love-letter [retrieved 06.07.2016].

[4] BoardGameGeek, Batman Love Letter, https://boardgamegeek.com/boardgame/168584/love-letter-batman [retrieved 30.06.2016].

[5] Alderac Entertainment Group, Archer Love Letter, https://www.alderac.com/loveletter/products/love-letter-archer/ [retrieved 30.06.2016].

[6] Alderac Entertainment Group, The Hobbit Love Letter, https://www.alderac.com/loveletter/products/hobbit-battle-five-armies/ [retrieved 30.06.2016].

[7] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis and S. Colton, A Survey of Monte Carlo Tree Search Methods, IEEE Transactions On Computational Intelligence and AI in Games, 4 (2012).

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, Third edition, The MIT Press (2009).