Universiteit Leiden
The Netherlands

Opleiding Informatica

A perfect information Scrabble game

Michel Klijn

Supervisors:
Rudy van Vliet & Jeannette de Graaf

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

08/08/2017

Abstract

Scrabble is a word game played by two to four persons. They place letters from their rack on a 15x15 board to create words. At least one new letter has to be placed next to a letter that is already on the board. Created words are awarded with points. The height depends on letter values and multipliers. The player with the highest end score wins.

This research is about a variant of Scrabble. There are only two players and all information, such as letters on the racks and in the bag, is accessible for both. This creates a whole new view on strategies to play this game. A Scrabble program has been written and several strategies are implemented: Random, Greedy, Greedy Look Ahead, Greedy with rack evaluation functions and Monte Carlo. All strategies have played against the other ones to find the best way to play this variant of Scrabble.

Contents

Chapter 1

Introduction

Scrabble [Has14] has been a popular word game for a long time. Alfred Mosher Butts invented a game called LEXICO back in 1933. This game has been refined twice to result in the current Scrabble game in 1948. Butts used his interest in puzzle games to create it and he analysed the American language to make a fair letter distribution. Scrabble is played on a board, but there are also digital versions nowadays.

Scrabble is played by two to four persons that have a chance to score points each turn. Letters from their rack are used to create words with the letters on the board. The score can be increased by using a hotspot that multiplies a letter or word. The score of a move is added to the player's score. The game ends when a rack is empty or all players passed after each other. The player with the highest end score wins the game.

Since there has been done a lot of good research on strategies for the standard Scrabble game, this research is about a variant with perfect information. The bag with new letters is accessible for all players, just like all letters on the racks are visible. Players can make smarter decisions when choosing a move, since they can predict their opponents reaction. Besides this visibility change, also some small changes in the rules are made to simplify the game for this research: only two players play the game and blank tiles are not used.

The most basic ways to play a game by a computer are Random and Greedy. The Greedy player determines all moves and picks the one with the highest score. Some other strategies will be designed that use the Greedy player as basis: Greedy Look Ahead and Greedy with a rack evaluation function. A rack evaluation function gives a score to the letters on a rack by analysing the frequency of letter combinations in the dictionary. The last strategy that will be designed is Monte Carlo, which is a combination of the Greedy and Random strategy.

In the end, all strategies play against the other strategies to see how each strategy performs.

Chapter 2

Related Work

As far as we know, there has not been done any research on our variant of Scrabble. However, a lot of research has been done on strategies for playing the original Scrabble game. One article worth mentioning is from Brian Sheppard [She02]. He had written a Scrabble program called Maven. He kept improving this program by simulating games and talking to human professionals on tournaments. Eventually, it is even able to win games against the best human players. He divides a game in three phases: middle-, pre-end- and endgame. The endgame starts when no tiles are left in the bag. At that moment, information about the opponnent's rack can be obtained. In the other phases, he uses a so-called 'oracle' to predict information about upcoming tiles from the bag and tiles on the opponent's rack. However, since the variant used in this research provides all information, this oracle idea will not be neccessary.

Brian Sheppard also explained the human approach of Scrabble, it can be summarized as follows:

- Look for a *bingo*: use all seven letters of your rack in one move to get bonus points.
- Look for *hotspots*: special squares that multiply a letter- or word score.
- Look at the *rack leave*: remaining letters on the rack after a move is done.
- Try to improve a found move.

We will use this information, in particular the rack leave, when designing strategies to play our variant of Scrabble.

Chapter 3

Game Rules

3.1 Materials and setup

The English version of Scrabble is played with 100 *tiles* in total, representing 27 different characters: 26 letters from the alphabet plus the blank tile. The blank tile can be used as any letter. Some letters are more common than others in a language, which means that those letters are more present in a Scrabble game. The distribution of letters over the tiles is presented in Table 3.1. This table also shows the value of each letter. More common letters have a lower point value.

This research is about a variant of Scrabble. All tiles, including tiles in the bag and on the opponent's rack, are visible during the game. The *bag* contains all tiles in the beginning. It is shuffled before the game starts and the tiles are put in one straight line. To fill racks, the first tiles of that line are the ones that should be used. Figure 3.1 shows a typical game setup of this Scrabble variant. The blank tiles are left out. In Table 3.2, a comparison is made between Scrabble and this variant.

The board is initially empty, but it shows the positions with a special meaning, also called *hotspots*:

- Start. The first word should cover this square. Its position is exactly in the middle of the 15x15 board.
- DL. This place doubles the value of the letter that is placed on this square.

Table 3.1: Distribution of letters.

Letter	Value	Amount	Letter	Value	Amount	Letter	Value	Amount
A	1	9	J	8	1	S	1	4
B	3	2	K	5	1	T	1	6
C	3	2	L	1	4	U	1	4
D	2	4	M	3	2	V	4	2
E	1	12	N	1	6	W	4	2
F	4	2	O	1	8	X	8	1
G	2	3	P	3	2	Y	4	2
H	4	2	Q	10	1	Z	10	1
I	1	9	R	1	6	Blank	0	2

Figure 3.1: Setup of game material.

Table 3.2: Comparison between the original Scrabble game and the used variant.

	Original Scrabble	Variant of Scrabble
Number of players	2-4	2
Number of letter tiles	98	98
Number of blank tiles	2	0
Bag	Not visible	Visible
Rack of opponent	Not visible	Visible

- TL. This place triples the value of the letter that is placed on this square.
- DW. This place doubles the complete word value when a letter is placed on this square.
- TW. This place triples the complete word value when a letter is placed on this square.

Figure 3.2 shows an empty board with hotspots. A hotspot loses its meaning when it is used once.

3.2 Game play

Each turn, a player creates words by putting letters of his rack on the board. Letters can be placed either horizontally or vertically. At least one letter should be placed next to a letter that was placed in a previous turn. If it is the first turn, a letter should be placed at the start hotspot. All words are awarded with points, where the height is dependent of letter values and the use of hotspots. If all seven letters from the rack are used in one move, it is called a *bingo*. This is rewarded with 50 extra points. Each turn ends with replenishing the rack with tiles from the bag, as long as the bag is not empty. Both players start with zero points. Their score is updated after each turn by adding the score of that turn.
It could happen that a player has bad tiles and is not able to place tiles on the board. In that case, he passes and the turn goes to the opponent. It is also allowed to do a strategic pass to avoid getting bad tiles from the bag or, in case the opponent already passed, to win the game based on the current scores. Strategic passing will not be used in our algorithms.

3.3 Goal and game end

The goal of this game is to achieve a higher end score than the opponent. A game ends when:

- The rack of a player is empty and can not be replenished with tiles from the bag.
- A player passes directly after his opponent passed.

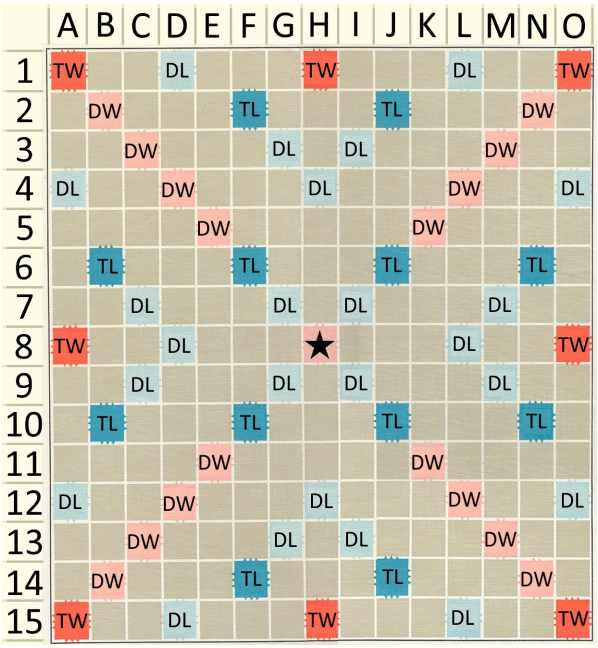

Figure 3.2: Hotspots: Start (star), DL, TL, DW, TW.

The value of the remaining tiles on the rack will be subtracted from the player's final score.

Chapter 4

Approach

4.1 Main program

For this research, a program in C++ was written to make the experiments possible. This section will describe some basic methods like checking if a word is valid according to the used word list. The program creates a Scrabble object containing all necessary variables and methods. The most important ones are listed below.

Variables:

- char bag[BAGWIDTH]. This array represents the bag. All tiles are in this array and they are shuffled before the game starts.
- char board[BOARDHEIGHT][BOARDWIDTH]. This 2d array is used to keep track of letters that are placed on the board.
- Hotspots hotspot[BOARDHEIGHT][BOARDWIDTH]. This 2d array contains the information about hotspots on the board. Every position has a value of the Hotspots enumeration: normal, dw, dl, tl, tw, used. If a square loses its special meaning, it will get marked as 'used'.
- int letterValues[ALPHABET_SIZE]. This array contains the value of each letter.
- Player 1 and Player 2. These objects contain information about the two players. For example the game strategy, rack and score.
- int distanceToSquareWithNeighbor[BOARDHEIGHT][BOARDWIDTH][2]. For every position and both directions, this array contains the distance to the nearest square that has a letter as neighbor. After a move is done, the program computes all new distances.
- bool validLetter[BOARDHEIGHT][BOARDWIDTH][ALPHABETSIZE][2]. This array gathers information about letters that have been tried before on a position with a direction (horizontal or vertical). Initially,

the array is completely true, which means that every letter needs to be placed to check if it is valid. When a letter is placed in one direction and the word formed in the orthogonal direction appears to be invalid, the letter on this position with direction is marked. This prevents checking that particular letter in that particular direction again next time.

Methods:

- void displayAll(). This will display the current game configuration, including the board, upcoming letters from the bag, the score and racks of the players.
- bool validWord(string word). Returns true if the given word exists in the dictionary (see Section 4.1.2).
- bool validMove(int i, int j, bool horizontal, string letters). Returns true if the move is valid. This method will be discussed in Section 4.1.4.
- void doMove(int i, int j, bool horizontal, string letters). This function gets a position (i,j). From there it will walk to the right or downwards, depending on the direction. If a position is empty, it will place the next letter from letters. Otherwise, it moves to the next position.
- void undoMove(int i, int j, bool horizontal). To check if a move is valid, letters are placed on the board and several checks are perfomed. After this validation, the move needs to be reversed, so that the next move can be evaluated on the same board. This method makes that possible by resetting only the positions that were changed by this move.

4.1.1 Definition of a move

A move consists of a start position, direction and letters to be placed on the board. The start position is a combination of a column and row. In the program, a boolean is used to indicate the direction. This can be either horizontal or vertical. Letters are placed on the board by iterating through the positions. If a position is empty, the program will place a letter, otherwise it just goes to the next position.

4.1.2 Word list representation

To check if a word is valid, a word list is needed. For this research, the *SOWPODS* word list is used. This English list is a combination of British and American words and is one of the most used dictionaries on Scrabble tournaments. It contains over 267.000 words, making it inefficient to walk through the whole list and compare a word with all those words. Therefore, the word list is represented as a trie (Figure 4.1). Starting at the root, every node contains information about whether it is a leaf node or not and it has 26 pointers to child nodes. The first pointer of a node represents the 'a', the second a 'b', and so on. To check if a word is valid, it only takes n steps, where n is the length of that word.

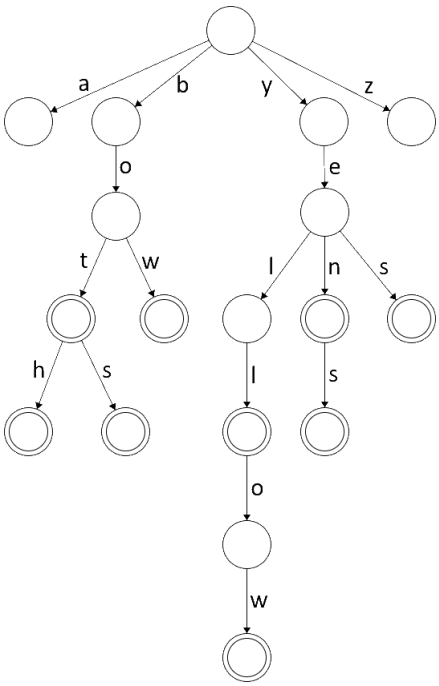

Figure 4.1: Part of the full trie, representing the following words: bot(s), both, bow, yell, yellow, yen(s), yes. Double circles indicate the end of a word.

4.1.3 Recursive creation of words

Later in this report, the players with their algorithms are described. A human considers a move by looking at the letters on his rack. A computer algorithm works differently. A recursive approach is chosen for this research. All letter combinations are created and added to a stack. A validity check is done afterwards.

4.1.4 Validity check of a move

The validMove method checks the validity of a move. Three checks are performed:

1. The first check is the length of the word. In some cases, letters might be placed outside the board, which is not allowed.
2. A word needs to be adjacent to another word or cover the start position in the middle.
3. The created word is checked by the validWord method, but it happens very often that a word is created in the orthogonal direction as well. Those words are also checked on their occurence in the word list.

If none of the above checks found problems, the method returns true. In all other cases, false is returned. The pseudocode of this method can be found in Algorithm 1.

Algorithm 1: validMove method.

```
if letters will be placed outside board then
    return false;
else
    initialize word with letters before startposition;
    while position is covered with tile or not all letters of move are placed yet do
        if position is covered with tile then
            add letter on tile to word;
            remember that an adjacent tile was found;
        else
            if letter has been marked as unvalid for this position then
                undo this move and return false;
            else
                place letter on position;
                add letter to word;
                if position is middle of board or has an adjacent tile then
                    remember that an adjacent tile was found;
                end
                if position has adjacent tile in orthogonal direction and forms unvalid word then
                    mark this letter as unvalid for this position;
                    undo this move and return false;
                end
                update current letter to the next letter of the move;
            end
        end
        update current position to next position;
    end
    undo move;
    check if word is valid;
    if word is valid and at least one adjacent tile then
        return true;
    else
        return false;
    end
end
```

4.1.5 Computing the score of a move

The score is calculated in the doMove method (see Algorithm 2) at the same time as placing tiles on the board. Inititally the score is 0 and the score multiplier is 1. Before placing tiles, the positions before the startposition are looked at. The sum of letter values of those tiles are added to the score. Then, the current position is evaluated. If there is already a tile on this place, we just increase the score with the letter value of that tile. If the current position is not used yet, we place a letter. The value of this letter is dependent on both letter value and hotspot. The letter value can be multiplied by 2, in case of DL, or 3, in case of TL. The score multiplier is mulitplied by 2, in case of DW, or 3, in case of TW. The score is still not correct, because the position can have neighbors in the ortogonal direction. After that score is calculated, it is added to the scoresInOtherDirections variable, which is initially 0. These steps are repeated until all letters are placed. The word score is multiplied with the score multiplier and then increased with the scoresInOtherDirections.

Algorithm 2: doMove method.

```
initialize word and score with letters before startposition;
while position is covered with tile or not all letters of move are played yet do
    if position is covered with tile then
        add letter on tile to word;
        add letter value to word score;
    else
        put letter on board;
        remove letter from rack;
        add letter to word;
        add letter value multiplied with letter multiplier of position to word score;
        multiply the word score multiplier with word multiplier of position;
        if word in orthogonal direction then
            add score of this word to the scores of all orthogonal formed words;
        end
        update current letter to the next letter of the move;
    end
    update current position to next position;
end
multiply word score with word multiplier;
add scores of orthogonal words to word score;
if bingo then
    add bingo bonus to word score;
end
add word score to the player's score;
replenish player's rack with letters from bag;
```

4.1.6 Finding all valid moves

Given a set of seven letters on the rack of a player, all combinations of those letters are put on a stack. This results in 13.699 combinations.

$$7! + \frac{7!}{1!} + \frac{7!}{2!} + \frac{7!}{3!} + \frac{7!}{4!} + \frac{7!}{5!} + \frac{7!}{6!} = 13.699$$

Then, every combination is placed on the board with every position as starting point and each direction and checked with the validMove method. This is quite a big job to compute, since there are 225 positions and two directions which results in $13.699 \times 225 \times 2 = 6.164.550$ times that validMove is called. That method performs a lot of checks and it is preferred to avoid these checks to reduce processing time. Therefore, some efficiency measures are built in.

- Eliminating double combinations. If a letter occurs more than once on a rack, it will produce some double combinations. The program prevents that a combination is added to the stack multiple times.
- Only free positions. Words are only placed from positions that are not covered yet with a tile. This reduces unneccessary duplicates.
- Current game area. The first and last row and column that contains a letter are saved and updated after each definite move. This action is very useful particularly in the beginning of the game. Figure 4.2 shows a game configuration after two moves. New moves are tried horizontally and vertically. Horizontal words are only checked between rows 5 and 13, vertical words only between columns D and J.
- Word length check. The program compares the word length and the number of consecutive squares without any occupied neighbor counting from the current position. If the word length is smaller than that number, it will not be adjacent to another word and thus not be valid.
- Valid letter check. For this check, the validLetter array is used. Some letters are already tried on a certain position with a certain direction. If they formed an invalid word in the orthogonal direction, it is not useful to check again.
- Prefix check. Besides checking the validity of a complete word, it can be efficient to check if the current placed letters are a possible prefix of a word. If not, the validity check of the current move can be stopped immediately since the outcome is clear: not a valid move.

4.2 Players and their algorithms

Several game styles are implemented. The first one is dependent on human input, the others have some strategy to automatically place words on the board. The implementations of all players are described in the next subsections.

4.2.1 Human

The player gives a startposition, a direction and the letters that he wants to place. With that information, the program determines the validity of the move. If the move is valid, it is played. Although this player is useful for testing the program's correctness, it will not be used for the experiments.

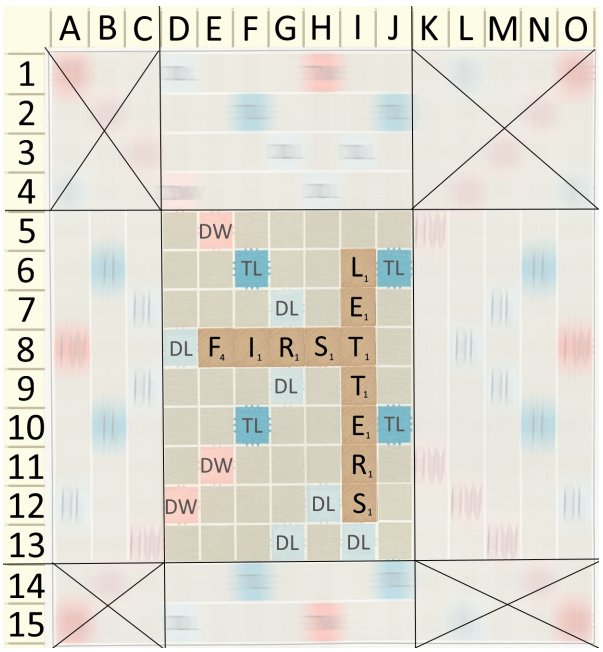

Figure 4.2: Current game area.

4.2.2 Random

This player does not have a good strategy, it just randomly chooses one of all valid moves. Therefore, it creates a list with all combinations with the letters on his rack. Then, a random combination from that list is tried for every position on the board, both horizontally and vertically. If a valid move is found, it is immediately the player's move for that turn. No other combinations need to be validated. If no valid move is found for this combination, the next combination on the list is tried until all combinations have been reviewed. If none of the combinations yield a valid move, the player passes. In Algorithm 3, the pseudocode of this strategy is written.

Algorithm 3: Random strategy.

```
make a list of all combinations of letters from rack;
choose a combination from the list randomly;
while not all combinations have been tried do
    foreach position p do
        if combination placed horizontally starting from p is valid then
            do move and return;
        else if combination placed vertically starting from p is valid then
            do move and return;
        end
    end
    move to next combination in list;
end
pass and return;
```

4.2.3 Greedy

The greedy player wants to play the word that yields the best score at that moment. The code can be seen in Algorithm 4. First, all combinations of letters on the rack are computed. For each of those combinations with each position and direction, the validity is checked. If the move is valid, the score will be compared to the current best score. If the score is higher, the move will be saved and the best score is updated. If the score equals the best score, the algorithm chooses the new move with a probability $\frac{1}{n+1}$, where n is the number of moves with the same score so far. The first move has a probability of 100% to be chosen, the second move has a probability of 50%, the third move 33%, the fourth move 25%, and so on. As a result, all moves with the same best score have the same probability to be chosen.

Algorithm 4: Greedy strategy.

```
make a list of all combinations of letters from rack;
foreach combination c do
    foreach position p do
        foreach direction d do
            place letters of c starting at p in direction d;
            check validity of move;
            if valid move then
                compute score of move;
                if score higher than best score then
                    remember this move and corresponding best score;
                else if score equals current best score then
                    choose one of these two moves with a fair probability distribution;
                end
            end
        end
    end
end
if no valid moves were found then
    pass and return;
else
    do move and return;
end
```

4.2.4 Rack strategy

This algorithm is very similar to the Greedy player in the basis. One of the moves with the best score will be made definite. However, the score does not only depend on the word that is placed, but also on some evaluation of the rack leave. The rack leave contains the letters from the rack minus the n letters used for the move plus n new letters from the bag. Research has been done to letter combinations and how to process them in an evaluation function.

Table 4.1: Most common combinations in SOWPODS word list.

L1	Occurence
E	69%
S	61%
I	59%
A	55%
R	52%
N	48%
T	47%
O	46%
L	39%
C	31%

L1L2	Occurence
ES	19%
IN	18%
ER	17%
TI	13%
TE	11%
AT	11%
IS	11%
NG	10%
ON	10%
RE	10%

L1_L2	Occurence
I_E	11%
A_I	11%
E_S	11%
I_G	10%
A_E	8%
E_T	8%
N_S	8%
O_E	7%
E_I	7%
O_I	7%

Combinations

A rack contains some letters that can form combinations. The program counted for each combination how many words in the used word list contain that combination. There are three kinds of combinations we searched for:

- L1 where there is only one letter. This results in the 'combinations' a, b, c, ..., y, z.
- L1L2 where L2 follows L1 directly. This results in the combinations aa, ab, ac, ..., zy, zz.
- L1_L2 where one letter separates L1 and L2. This results in the combinations a_a, a_b, a_c, ..., z_y, z_z.

All words in the word list are looked at to see if it holds one of the combinations. The most common combinations are shown in Table 4.1. An occurence of 69% of the 'E' indicates that 69% of all words contain at least one 'E'.

Evaluation function

The information of the occurence of combinations is used to create an evaluation function. Several formulas have been applied, trying to give all combinations an appropriate value. Table 4.2 shows the ten best valued combinations for each of the following formulas:

1. $L1 \times V1$, $L1L2 \times (V1 + V2)$, $L1\_L2 \times (V1 + V2)$. V1 and V2 are the letter values of L1 and L2. These formulas combines occurence (Table 4.1) and letter values (Table 3.1).
2. $L1$, $\frac{L1L2}{L1+L2}$, $\frac{L1\_L2}{L1+L2}$. These formulas relate the frequency of a combination to that of the individual letters in the combination. We can see in Table 4.2 that the combinations IN, I_G and NG have good scores with this formula. Together they form a three letter combination ING, which is frequent in English. This example shows that two-letter combinations as used in this research also represent three-letter combinations. We do not use the letter values here, because we want to focus on the ease to play out a certain combination. Besides, it is not desired to hold tiles with a high letter value till the end of the game, since it might result in deduction of points.

3. $L1$, $\frac{L1L2}{min(L1,L2)}$, $\frac{L1\_L2}{min(L1,L2)}$. min(a,b) returns the smallest number of the two parameters. These formulas show the dependency of a combination of the least frequent letter. A very interesting example is the combination QU, which is also mentioned in the article about Maven [Sheo2]. In our word list, there are 4069 words with a Q, of which 3985 words have the combination QU. Playing a word with a U when there is a Q on the rack could result in a penalty when the game ends if the Q is still on the rack. We do not use the letter values here for the same reason as mentioned above for the second formulas.

We will use the last set of formulas in this research, because it highlights the combinations we want to be highlighted (see Table 4.2). To continue with the example of the combination QU, a rack with a Q is not desired, since it is relatively hard to play a word with it. Therefore, it is only rewarded with 1 point compared to 69 for a single E. If there is also a U on the rack, however, it is a very strong combination and useful for a future move.
The weights of L1, L1L2 and L1_L2 in a rack should be optimal. For this, different weights for these variables have been tested while playing against the Greedy player. The ratio of a move score to a rack score has also been tested with different weights. Further information and results are given in Chapter 5.
A possible negative effect of this evaluation function is getting stuck with the more common letters and combinations, like E's. For example, a rack containing five E's may be undesirable. A combination limit has been implemented to avoid this. This limit has two variants:

- Simple. All combinations are only awarded with points once.
- Advanced. Some combinations are more useful than others. For example, QU is useful, but QUU might be getting too much points, causing the retention of both U's. On the other hand, having two E's is not that bad. We created a formula that is used for each combination on the rack. It uses two variables: #onrack indicates how often the combination can be formed with the letters on the rack, the relative score of a combination L1L2 is the value of L1L2 according to the third set of formulas. Some relative scores can be seen in Table 4.2. We want to count combinations with a high relative score only once, but combinations with a lower relative score may be given points multiple times, in such a way that $1 <= limit <= \#onrack$. This results in the following formula: $limit = \frac{\#onrack+relativescore}{1+relativescore}$.

4.2.5 Greedy Look Ahead

Presumably, the Greedy player will be hard to defeat since it knows all words and plays words with the highest possible score of that moment. But a Greedy player does not take the rack of its opponent into account. Sometimes it is better to play a word with a slightly lower score if that means that the opponent will get an even lower score in his next turn. The average results are eventually most important. To make this possible, a new player is introduced. It is a Greedy player and assumes that its opponent also is a Greedy player. Just as normal, it computes all possible moves and their scores. But, instead of playing the best score, it works out the most promising moves, i.e., the moves with the highest scores. After playing a move, it computes what his opponent will do. The scores are subtracted. After working out all the moves, the move with the best resulting

Table 4.2: Best combinations per formula.

Formulas 1							
L1	Points		L1L2	Points		L1_L2	Points
C	94		ES	38		I_G	28
H	83		IN	35		I_E	22
P	72		ER	34		A_I	21
M	70		CH	33		E_S	21
E	69		NG	31		M_N	17
S	61		IC	29		A_E	16
I	59		ED	28		E_T	16
Y	56		IZ	27		N_S	16
A	55		TI	25		P_R	15
D	54		HE	24		O_E	14

Formulas 2							
L1	Points		L1L2	Points		L1_L2	Points
E	69		IN	16		I_G	11
S	61		NG	14		A_I	9
I	59		ES	14		I_E	8
A	55		TI	14		E_S	8
R	52		ON	11		N_S	7
N	48		AT	10		E_T	7
T	47		ED	9		T_R	6
O	46		AL	9		A_E	6
L	39		CH	9		O_I	6
C	31		TE	9		O_E	6

Formulas 3							
L1	Points		L1L2	Points		L1_L2	Points
E	69		QU	97		I_G	42
S	61		IZ	60		Q_I	36
I	59		VE	51		Q_A	29
A	55		EX	50		Q_E	27
R	52		ZE	46		V_R	26
N	48		NG	46		A_I	20
T	47		IN	36		A_K	19
O	46		ED	35		I_E	18
L	39		ER	33		M_N	18
C	31		ES	31		L_Z	18

score is played. The number of most promising moves to work out is detemined in Chapter 5.

4.2.6 Monte Carlo

It is never absolutely clear what move the opponent will do, but it can still be useful to think some moves ahead. This Monte Carlo player computes the best scoring moves of that moment and will play a few turns ahead for each of them using the random algorithm for both players. The end scores of the players are subtracted after which the best end score determines which move should be done. Experiments in Chapter 5 have been done to optimize this algorithm's variables.

Chapter 5

Experiments

All experiments and their results are discussed in this chapter. First, the values of the variables for some strategies are explained. Then, all strategies play against all strategies to measure their performances.

5.1 Strategy variables

In Chapter 4, all strategies have been explained. However, the values of some variables can have a huge influence on the performance of a strategy. To get good values, a strategy plays against the Greedy player, with different values for its variables. The values yielding the best results are taken as the default values for that strategy. The Greedy player was chosen as the opponent, since it is a very basic strategy that performs already really well.

5.1.1 Rack evaluation function

The rack evaluation function comes in three versions: basic, simple limit and advanced limit. They use the same ratio for the weights of the different types of letter combinations. This ratio has been established by experimenting with different weights of L1, L1L2 and L1_L2. We created some 'good' racks, some 'bad' racks and some average racks, six racks in total. A good rack can contain letters like 'EGINRST', because these letters can form some of the most common combinations in the used word list (see Table 4.1). A bad rack can be 'BCJKQXZ' and an average rack can be 'AAELNNW'. Having a list of racks, ordered by the expected rack score, all racks were given a score using different weights for L1, L1L2 and L1_L2. Each rack score is divided by the sum of all weights. This resulted in lists with different orders. We then selected the weights that yielded an order that was most similar to the desired order. This resulted in 1:6:2 for L1:L1L2:L1_L2. Using the basic evaluation function and these values, our racks get the following scores: EGINRST (437), AAELNNW (341) and BCJKQXZ (44).
The rack evaluation function gives a score of the rack after a move is done. The move also has a direct score.

Table 5.1: Results of variating the number of best moves for the Greedy Look Ahead player.

# Matches	Player 1	Player 2	n best moves	Wins player 1	Wins player 2	Score difference
100	Greedy	Look Ahead	2	46%	54%	2
100	Greedy	Look Ahead	4	48%	51%	-1
100	Greedy	Look Ahead	6	40%	58%	-7
100	Greedy	Look Ahead	8	46%	53%	-10
100	Greedy	Look Ahead	10	44%	55%	-15
10	Greedy	Look Ahead	20	40%	60%	-23

Both scores are added to get a final score to compare the move to other moves. However, a rack score is usually a lot higher than a move score. A typical move score is 30 points, whereas rack scores can be more than 500. There has to be some balancing to obtain a fair final score. In the end, the current score turned out to be most important, but the rack may have a good influence. Our experiments led to the following weighings of move score and rack score.
The Basic Rack strategy multiplies the move score by 75 and then adds the rack score.
The Simple Limit Rack strategy counts every combination only one time, so the rack score is lower. The move score is therefore multiplied with only 25 before the rack score is added.
The Advanced Limit Rack strategy has rack scores somewhere between the rack scores of Basic and Simple Limit, so the final score is calculated as 50 times the move score plus the rack score.

5.1.2 Greedy Look Ahead

The Greedy Look Ahead player has one variable, namely the number of best moves that must be worked out for the opponent: n. So, to determine a good value for n, the Greedy player plays agains this Greedy Look Ahead player with different values for n. The results can be seen in Table 5.1. The Look Ahead player needs more time when n is higher, since it has to compute the best move for the opponent after each of these n moves are done. The most interesting thing that this experiment shows, is the score difference. Although the number of wins does not seem to be clearly related to n, the score difference does. The score difference is computed by subtracting the average end scores of player 1 and player 2. The results show that the average score of the Look Ahead player grows more than the average score of the Greedy player when n increases. Ideally, the best value of n is just as high as possible, so that all moves are processed. Unfortunately, this is not feasible because it requires too much computing time. The value we chose for n is 10, because it yields a good performance and a reasonable computing time for the Look Ahead player.

5.1.3 Monte Carlo

The Monte Carlo strategy uses three variables:

- n as the number of best moves for the current player.
- d as depth. Each of the n moves will be worked out with a total of d random moves, in such a way that both players have $\frac{d}{2}$ random moves to do.

- r as the number of random games played after each of the n best moves.

This strategy can result in large processing times, so there has to be a trade off between time and performance. r needs to be a high number to make the average score reliable. However, if both r and d are high numbers, the processing time will grow rapidly. We experimented with the Monte Carlo player with $n = 10$, $r = 100$ and different values for d, against a Greedy player. Results seemed to get better with smaller values for d. This might be explained by the fact that a game configuration changes by every move that is done. There is an enormous list of valid moves for each configuration. Picking one move of that list randomly is not a good indication of the move that the player would really choose. This does however change the game configuration and with that the next random move. The higher the depth, the less realistic the random games represent a game progress. So, the values we expect to work best are $r = 100$ and $d = 2$. To find a reasonable value for n, games were played between Greedy and Monte Carlo with values varying from 1 to 10. Taking time into account, more than 4 best moves did not provide a significant improvement. The final values of the Monte Carlo strategy that are used in the games in Section 5.2 are: $n = 4, d = 2, r = 100$.

5.2 Game results

All strategies have been designed and optimised with good values for the variables. Now, they will compete against each other. Every strategy starts a game against all strategies, including itself. For example, there is a game where the Random player starts against the Greedy player, but there is also a game between these two players where the Greedy player starts doing a move. This results in $7 \times 7 = 49$ different games. The results are summarized in Table 5.2, Table 5.3 and Table 5.4. All games have been played at least 1500 times to make the results reliable. Winning a game depends on the used game strategies, but also on the letter distribution on the racks and the letter sequence of the bag. To determine whether the results are statistically significant, we look at the chance of the starting player to win a game. Suppose that both players have a probability of 50% to win a single game: $p = 0.5$. Since we played each game $n = 1500$ times, we can calculate the 95% confidence interval [AC98]. First, we calculate the standard deviation: $\sigma = \sqrt{\frac{p \times (1-p)}{n}} \approx 0.0129$. Then we can calculate the lower and upper bounds using the critical value $z = 1.96$. The lower bound is calculated as $p - z \times \sigma = 0.5 - 1.96 \times 0.0129 \approx 0.474$ and the upper bound is calculated as $p + z \times \sigma = 0.5 + 1.96 \times 0.0129 \approx 0.526$. This means that, if 1500 games are played between two players with $p = 0.5$, we are 95% confident that player 1 wins 47.4% to 52.6% of those games. Results in Table 5.2 show a lot of winning rates outside these boundaries, which means that the results not just depend on chance, but mostly on the used strategies.

Advantage of the starting player

Looking at the winning rate (Table 5.2) and score differences (Table 5.3) of a player playing against another player with the same strategy, it is clearly important to be the starting player. All starting players, except the random player, have a winning rate over 60%. However, the game between Random and Random has been

Table 5.2: Winning rate of starting player 1 (left) against player 2.

	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Random (1)	52%	0%	0%	0%	0%	0%	0%
Greedy (2)	100%	63%	59%	47%	58%	38%	67%
Rack Basic (3)	100%	65%	60%	51%	60%	45%	71%
Rack Simple Limit (4)	100%	76%	70%	63%	69%	54%	78%
Rack Advanced Limit (5)	100%	67%	62%	50%	62%	43%	69%
Greedy Look Ahead (6)	100%	84%	82%	72%	80%	65%	85%
Monte Carlo (7)	100%	54%	54%	42%	51%	38%	62%

Table 5.3: Average score differences between starting player 1 and player 2.

	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Random (1)	3	-341	-352	-371	-350	-333	-355
Greedy (2)	357	27	18	-7	17	-22	35
Rack Basic (3)	362	29	23	3	22	-12	42
Rack Simple Limit (4)	390	54	46	26	44	9	63
Rack Advanced Limit (5)	371	35	28	3	25	-12	44
Greedy Look Ahead (6)	339	65	58	39	58	24	72
Monte Carlo (7)	369	9	9	-18	7	-25	24

Table 5.4: Average scores of starting player 1 against player 2.

	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Random (1)	225	135	132	133	134	132	124
Greedy (2)	485	376	373	367	375	335	380
Rack Basic (3)	491	377	372	375	378	343	386
Rack Simple Limit (4)	516	399	400	393	398	360	405
Rack Advanced Limit (5)	497	382	377	377	379	343	385
Greedy Look Ahead (6)	458	358	356	360	366	328	365
Monte Carlo (7)	486	369	368	364	371	333	373

Table 5.5: Average results of the starting player 1 against player 2.

	Winning rate	Average score
Player 1	59%	353
Player 2	41%	334

Table 5.6: Average results per player.

Player	Winning rate	Average score	Average score difference
Random	0%	128	-358
Greedy	52%	374	46
Rack Basic	55%	379	53
Rack Simple Limit	65%	400	80
Rack Advanced Limit	56%	382	56
Greedy Look Ahead	73%	375	86
Monte Carlo	47%	370	38

played 215.000 times, so this small advantage of 52% is still significant evidence of the starting advantage. Besides the winning rate, also the score differences are in advantage of the starting player. Table 5.5 contains the average overall results of Table 5.2 and Table 5.3 and it shows that the starting player wins 59% of all its games against all strategies, with an average score difference of $353 - 334 = 19$ points.

Bad influence of Random on Monte Carlo

It is obvious that the Random player loses almost all of its games, since his choices are based on chance. Table 5.6 shows the combined results of all games per strategy. Since both games of player 1 against player 2 and games of player 2 against player 1 are used to create this table, the starting advantage does not affect these results. Results of games between players with the same strategy are not used. If the Random strategy plays against all other strategies, it wins 0% of the time with an average score of 128. His opponents score on average 358 points more. Because of the low winning rate of the Random player, most of the other players have a winning rate above 50%.
Monte Carlo picks the best greedy moves and works all of them out by playing a lot of random games. This was expected to improve the Greedy player at least a little bit. However, the results are very disappointing as can be seen in Table 5.6. Apparently, the random games cause the Monte Carlo player to choose the wrong moves. It was developed to make a somemore defensive choice, but its score difference is even lower than the Greedy player.

Best scoring strategies

The strategies with the highest average end score are the ones that base their moves on both move score and rack leave score (see Table 5.6). The Greedy player has an average of 374 points, while the Rack Simple Limit player scores 400 points per game. The ideal opponent of the Rack Simple Limit player is the Random player. The average score is 516 points when the rack player starts the game (see Table 5.4). The results of the rack strategies show that playing the moves with the highest scores as the Greedy player does, does not lead to the

highest end scores. A good rack evaluation function helps getting higher scores for future moves and also a better score difference. Table 5.6 indicates that an evaluation function that counts every combination on a rack only once works best. A reason for this may be that such a function yields a greater variety of the letters on the rack, which gives more opportunities to create words in a next move. However, it is possible that other values of the variables would lead to other, possibly better, results.

Best winning strategies

Getting a high score leads to good results, but not the best winning rate. Table 5.6 shows that, on average, the Greedy Look Ahead player scores $400 - 375 = 25$ points less per game than the Rack Simple Limit player. However, the score difference is in advantage of the Greedy Look Ahead player. And even more importantly, this player is the one winning most, with a winning rate of 73%. The results of this defensive strategy are clear: find a good move for yourself, but ensure that the opponent can not play a good move in response. The most competitive strategy is the Rack Simple Limit player. If the Greedy Look Ahead player starts the game against the Rack Simple Limit player, it wins 72% of the time, but the Rack Simple Limit player wins 54% of the games when it starts against the Greedy Look Ahead player (see Table 5.2). So overall, the Greedy Look Ahead player wins 59% of the games against the second best strategy used for this research. Expecting the opponent to be a Greedy player has shown to be a great approach to play and win this variant of Scrabble.

Chapter 6

Conclusions and further research

There is not one way to play this perfect information Scrabble game. However, two approaches have shown impressive scores and winning rates: Rack Simple Limit and Greedy Look Ahead. The results have shown that only doing the best move each turn, as the Greedy player does, is not enough to win from more 'intelligent' strategies.

Taking the rack leave into account ensures a better base for the next move. Information about common combinations in a word list were used to value all possible combinations. Bad combinations on a rack are averted, while useful combinations are favored on the rack. Using a good ratio between the move score and rack score prevents that the rack gets too much influence.

Besides the rack leave, it is also very useful to look at the opportunities for the opponent. If a certain move exposes a hotspot that can results in a high score for the opponent, it could be smarter to do a move with a slightly lower score. Doing so in such a way that the score difference between a player and its opponent is as high as possible, results in a high winning rate.

The main difference between Scrabble and this variant is the availabilty of information. Players can use information about the bag and the opponent to make smarter decisions. The rack strategies use information about which new letters a player gets after a move is done. Since the Rack Simple Limit player has a better winning rate than the Greedy player, it is indeed useful to have information about the bag. The same applies to the letters on the opponent's rack. The Greedy Look Ahead uses this information to predict the opponent's reaction to its own moves.

To conclude, this perfect information Scrabble game makes approaches possible that use information about the bag and the opponent's rack. An evaluation function for the rack leave has a positive influence on the score and by preventing the opponent from making high scoring moves, the winning rate will be even higher.

It might be very interesting to implement a new strategy that combines the Greedy Look Ahead player and the Rack Simple Limit player. In that case, the best n moves according to the Rack Simple Limit algorithm are listed. All of those n moves are worked out by finding the best move of the opponent, assuming that he uses the Rack Simple Limit strategy. With such a strategy, you focus on your own score and rack leave, but also on the opponent's score and rack leave.

Bibliography

[AC98] Alan Agresti and Brent A. Coull. Approximate is better than exact for interval estimation of binomial proportions. *The American Statistician*, 52(2):119–126, 1998.

[Has14] Hasbro. About the game. `https://scrabble.hasbro.com/en-us/history`, 2014.

[She02] Brian Sheppard. World-championship-caliber Scrabble*. *Artif. Intell.*, 134(1-2):241–275, 2002.