Automatic Wordfeud Playing Bot

Authors:
Martin Berntsson, Körsbärsvägen 4 C, 073-6962240, mbernt@kth.se
Fredric Ericsson, Adolf Lemons väg 33, 073-4224662, fericss@kth.se

Course:
Degree Project in Computer Science, First Level
DD143X
CSC
KTH

Supervisor:
Johan Boye

Abstract

Wordfeud is a version of the board game Scrabble adapted to smartphones.
In this report we describe the algorithm for the implementation of a greedy wordfeud playing bot and evaluate its performance (time and score) against a random bot and itself. The average time for calculating the move with the most points was 39.7 milliseconds. In our results the greedy bot always wins against the random bot with an average 600 point score per game against the random bots average 150 points score per game.
We draw the conclusion that our greedy bot has a viable strategy for playing the Wordfeud game.

Table of contents

1. Purpose

The goal of this project is to create a automatic wordfeud playing bot that will play the game generating as many points possible and thus winning the game. To do this project a algorithms that instructs the bot on how it should play the game will be created. A goal is to make this algorithm as fast as possible.
A bot that plays the game completely randomly will also be implemented to be used during the testing phase to measure the difference in the amount of points generated during a game. This will show us if our algorithms strategy for playing the game is a viable one.

Later, our bot could be used as some kind of cheating application or use it as some form of single player game where you play against the bot.

2. Background

2.1 History

Wordfeud is a relatively new game only available for smartphones such as android, IOS or windows phone. The game is based on the board game commonly known as “Scrabble” which is a game created during the early 19th century [3].
Released in 2010 wordfeud grew popular very quickly and today it has 10-50 million installs [2].

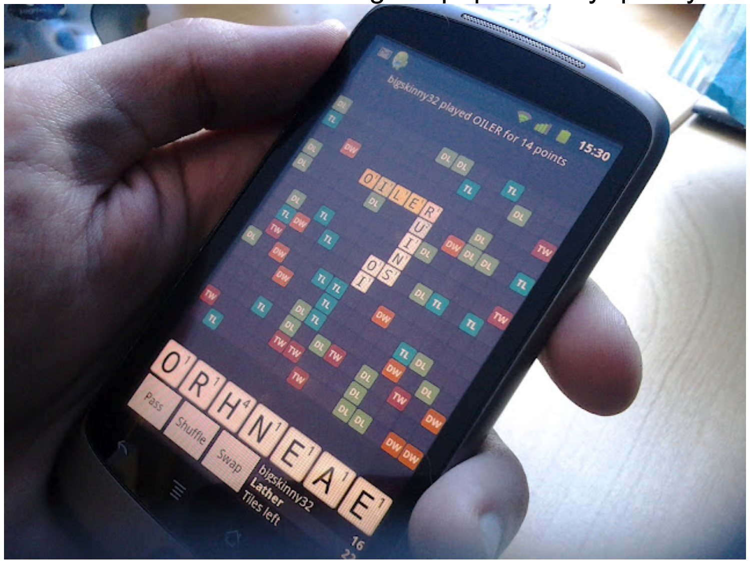


Figure 2.1, An example of how the game looks on a smartphone.

2.2 Game mechanics

Wordfeud is played by two players each taking turns at creating words on the game board. The goal is to create words that provide you with most amount of points. The person with the greatest amount of points at the end of the game is the winner.

2.2.1 Tile

The letters that you use to create words in the game are called tiles. This is because in the board game called scrabble you use white tiles with the letters printed on their faces to play the game.

2.2.2 Game cache

In scrabble a bag is used to contain all the tiles at the beginning of the game. In wordfeud there is no name for this bag, so in this report we choose to call it game cache. There are 104 tiles in the game cache at the start of the game. As seen in Figure 2.3 there are 102 normal letter tiles and 2 wildcard (blank) tiles in the game.

2.2.3 Rack

Each player has a rack consisting for 7 tiles that they may use during the game to construct words. Each player is assigned 7 at the start of the game leaving 90 left in the game cache. The two players take turns at creating and placing words on the 15x15 game board using the 7 tiles in their rack (See the bottom row of letters in figure 2.1) and other tiles on the board. When a player has made his move his rack is refilled taking random tiles from the games cache so that the player always has 7 tiles in his rack. If the game has no more tiles to provide the player will have less then 7 tiles in his rack.

2.2.4 Valid move

The tiles must be placed in a horizontal row or a vertical column with (no free spaces on the board between the first and last placed tiles). At Least one letter must be placed on the board to be a valid move or the player passes. At least one placed tile must be adjacent to a tile already on the board. All new words created from placing these tiles must be valid words (must be in the wordlist). Valid words can either be read from left to right or from top to bottom.

2.2.5 Passing

If a player passes, he forfeits his turn and allows the opponent to take his turn. If there are 3 passes in a row the game ends and the winner will be the person with the most amount of points.

2.2.6 Swapping tiles

If it’s your turn to play you have the option of switching some or all of your tiles with new random ones from the game cache, if there are less then 7 tiles in the game cache you are not allowed to swap tiles. After switching tiles your opponent gets to play.

2.2.7 game board

The game board consists of a 15x15 grid where you can place tiles.
When you start a new game the first move must be made with a tile placed at the center (7,7).
The game board has several bonus tiles spread out either randomly (see Figure 2.1) or in a default set manner (see Figure 2.2).
These bonuses consists of:

- DL (double letter): The points for the tile placed on this square is doubled.
- TL (triple letter): The points for the tile placed on this square is tripled.
- DW (double word): The points for the word that crosses this square is doubled.
- TW (triple word): The points for the word that crosses this square is tripled.

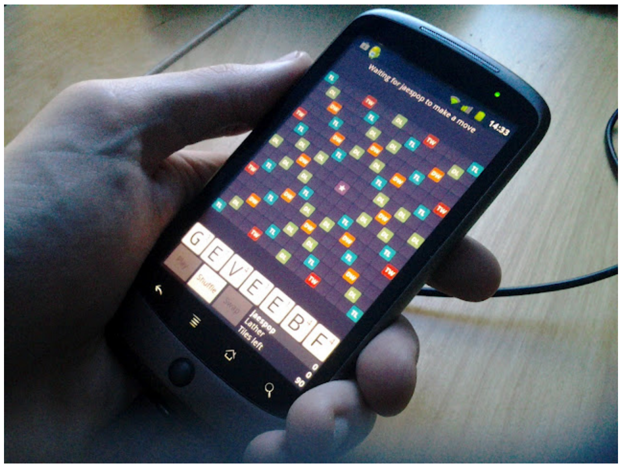


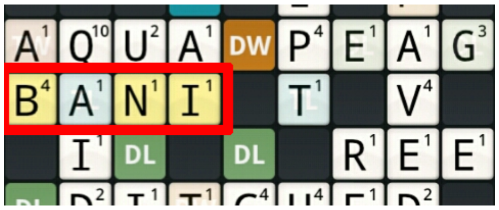
Figure 2.2, New default game

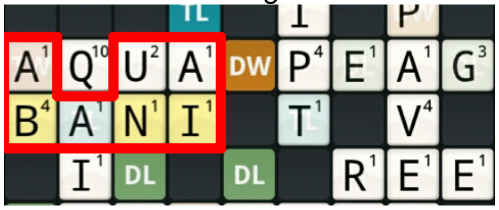
Note that if you create several words intersecting the tile DW or TW all of those words points are affected.

2.2.8 Points:

You get points for all words that were newly created on the board during your turn. Each tile give a certain amount of points (each tile type has a value). Newly placed tiles on bonus squares give bonus to all words they are parts of (those already on the board give no bonus). Blank tiles give zero points but they still have the same bonus rules.
If a player places 7 tiles, he receives an extra 40 points.

For example if you place the tiles B,N and I like this:


Playing these 3 (the yellow ones) tiles seen in the figure above will give you points for all these tiles seen in the image below:



English

Letter	Count	Points	Letter	Count	Points
A	10	1	N	6	1
B	2	4	O	7	1
C	2	4	P	2	4
D	5	2	Q	1	10
E	12	1	R	6	1
F	2	4	S	5	1
G	3	3	T	7	1
H	3	4	U	4	2
I	9	1	V	2	4
J	1	10	W	2	4
K	1	5	X	1	8
L	4	1	Y	2	4
M	2	3	Z	1	10

Figure 2.3 List of letters, their points[1] and the amount of them in the game cache

2.3 Previous work

2.3.1 Scrabble

MAVEN is one of the better (or best) scrabble playing bots [5]. It uses several techniques to get good results. It uses a data structure called DAWG [4] to quickly find all possible moves. For each move it uses a Monte Carlo [10] method to repeatedly simulate a few moves into the future (with random racks) and uses the average to adjust the points of each move. It also uses heuristics to determine how good a rack is and how bad or good it is to “open up” board positions, with some of the parameters determined through simulations. When nearing the end of the game it’s uses different simulation parameters, such as for example simulating to until the game is over. When there a no tiles left in the bag (and therefore no randomness) it uses a B* [11] search algorithm to find an endgame that is close to optimal (the specific algorithm used is described in [5]).

MAVEN has been developed intermittently for over 10 years (1986-2001). So its performance (speed and skill) should be superior to anything we could make.

2.3.2 Wordfeud

Mastermind [9] is a “cheating/assistant” applications for wordfeud for android.
“Wordfeud Mastermind logs in to your account and lists every possible move sorted by score. For all your games. For all your boards. For all your languages. No manual input is required by this app!”.

We don’t know what algorithm they use for finding all possible moves. But it should give the same moves as our greedy bot.

3.1 Programming Language

We have chosen create this software in Java using Eclipse as our programming environment. The reason behind this choice is that we both find Java to be the language that we are best at.

3.2 Communication with wordfeud servers

We found a wordfeud client for PC made in python[7] that communicates with the wordfeud servers using a http protocol. By using the http protocol we can now retrieve information about all the current games on an account and thus automatically calculate what all the moves can be made in all the different games.

3.2.1 Server protocol

The protocol is a rather simple http[6] connection where information being sent is put in the data section of the connection.
For example the information sent when logging in can look like this:

```
Content-type   application/json
Host           game03.wordfeud.com
Connection     Keep-Alive
User-Agent     WebFeudClient/1.2.8 (Android 2.2.3)

{"password": "3a243d74b56cf345c7bcd1f5596a712e4fb448dc","email": "email@gmail.com"}
```

Only thing the application needs to do is send a http GET/POST requests to certain URLs with headers and data attached.
Data sent and retrieved are in the JSON[8] string format.

3.5 greedy bot

The core in our software will be the greedy algorithm that given a game board and the rack (The tiles that the player can place) then calculates all possible moves that the player can currently make.

Our implementation of a greedy bot uses cascading filters to minimize the amount of calculations that are needed for each word.

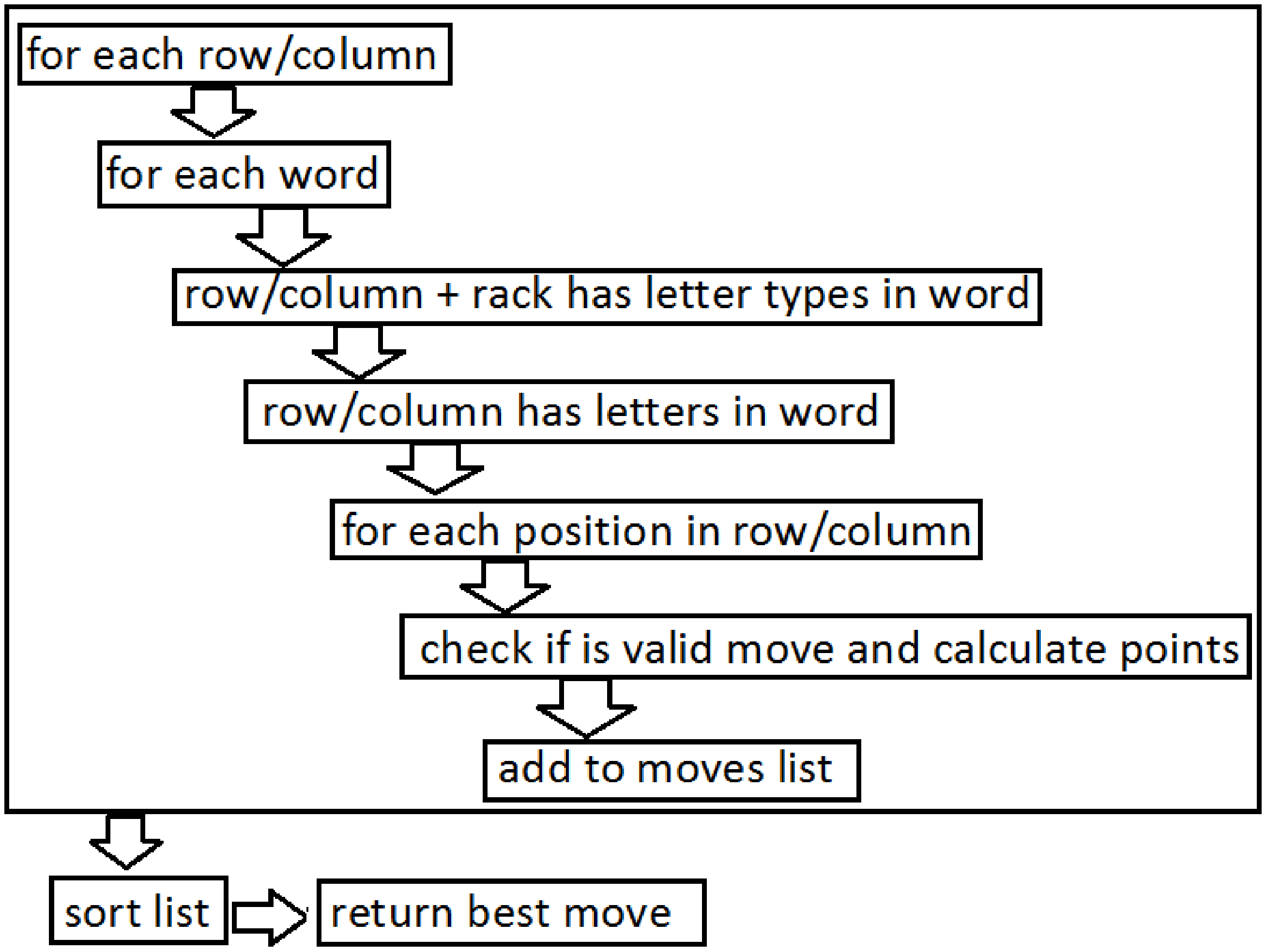

for each row/column
for each word
row/column + rack has letter types in word
row/column has letters in word
for each position in row/column
check if is valid move and calculate points
add to moves list
sort list
return best move

Figure 3.1 Illustrating the use of cascading filters.

The greedy bot uses the method called getGreedyMove. As can be seen in the pseudo code (see code 3.1) before trying to fit words to a position it first filters the words on what words can be built from the rack and the row/column (the filtering method is described in section 3.6 Filtering method). Then it uses the remaining words to try to fit them to each position and if they fits it adds the move (the move is an object with a coordinate, a word, a direction and the points for the move) to a list of moves. Finally it returns the move with the most points.

Code 3.1: getGreedyMove
Gets the move with the most points.

```
For each row/column:
    Filter wordlist with the letters in row/column and in the rack //code 3.4
    For each word in filtered wordlist:
        For each position on row/column:
            Check if it’s a valid move to place the word on position.//see code 3.2
            If it’s a valid move:
                //for points calculation see code 3.3
                create a move object with the points for the move included
                add move object to a list of valid moves
Finally sort the list of valid moves on points and return the move with the most points.
```

Code 3.2: isValidMove
Checks if a word can be placed at a position

```
if has letter before or after word:
    return false
if has no adjacent letter and isn’t on a letter:
    return false
if has corresponding letter on row that isn’t the same as in word:
    return false
if has at least one crossing word that is not in the wordlist:
    return false
if can construct word from letters in rack and letters under word:
    return true
else:
    return false
```

Code 3.3: points
Calculates the points for placing a word at a position with a given direction (horizontal or vertical).

```
# start values
totalCrossPoints = 0   # the total points for all the new crossing words
wordPoints = 0         # the points for all the letters in the word
wordBonusFactor = 1 # this is the total wordbonus that will be applied to the word
usedLetters = 0        # used to check if the bonus for using 7 letters should be applied

# loop
for each position pos1 in word and corresponding pos2 on board:
    letterPoints = points(word[pos1])
    letterBonus = 1
    wordBonus = 1
    if row[pos2] == '  ':
        usedLetters ++                      # count the number of used letters from rack
        letterBonus = letterBonus(pos2) # check if there is a bonus on the board
        wordBonus = wordBonus(pos2) # check if there is a bonus on the board
        if there is a crossing word:
            crossPoints = 0
            for each letter in crossing word:
                if letter is on the same position as pos1:
                    crossPoints += letterPoints * letterBonus
                else:
                    crossPoints += points(letter)
            totalCrossPoints += crossPoints * wordBonus
    wordPoints += letterPoints * letterBonus
    wordBonusFactor *= wordBonus

# if 7 letters are used from the rack you get a bonus of 40 points
if usedLetters == 7:
    bingo = 40
else:
    bingo = 0

# return the total points
return  wordPoints * wordBonusFactor + crossPoints + bingo
```

3.6 Filtering method

3.6.1 FastFilter

FastFilter is first initialized/constructed (see code 3.4.1) by counting the letter frequencies (i.e. number of letters of each letter type) type for each word in the wordlist and as an extra optimization for each word the needed letter types is embedded into an int. These calculations need only be done once because they will always give the same result for the same wordlist (and the word list doesn't change).

Once the FastFilter has been initialized/constructed the filtering (see code 3.4.2) works in the following way for the given row/column:
The number of blank tiles in the row/column are counted.

All the letters in the rack (except the blank tiles) and all the letters on the row/column are copied to a string. Then an integer with the letters types (in the string) embedded is made and the letter frequencies of this string are calculated. The previous operations described in this paragraph is fast because they are only done once per call to this filtering method. Then for each word in the wordlist, check if the word has all the letter types and then if there’s enough of each letter type and if so it’s added to a list of possible words that can be placed on the given row/column.

The check for letter types is much faster than the check for letter frequencies , because the first has constant time complexity and the latter has a time complexity depending on the the number of letter types in the word/string.

Code 3.4: FastFilter
Used to return a list of words with words that are impossible to construct with the letters from the rack and the row/column filtered out.

Code 3.4.1: Initialization/construction:

```
letterFreqs= two dimensional array of bytes
checkList= two dimensional array of bytes
letterTypes= array of ints
i=0
for each word in wordlist:
    letterFreqs[i]=countFrequencyOfEachLetterType(word)//code 3.5
    checkList[i]=getListOfIndexesContainingValuesGreaterThanZero(letterFreq)
    letterTypes[i]=letterTypesInWord(word)//code 3.6
    i=i+1
```

Code 3.4.2: Filter:

```
//do this for a given row/column
//lettersOnRow can also be lettersOnColumn
def String[] filter(String rack,String lettersOnRow):
    blanks=count blanks in rack
    rack2=remove blanks from rack
    letters=get all the letters in rack2 and the current row/column
    hasLetters=countFrequencyOfEachLetterType(letters)//code 3.5
    hasLetterTypes=letterTypesInWord(letters))//code 3.6
    res=[ ]
    i=0
    for each word in wordlist:
        if blanks>0 or hasLetterTypes(letterTypes[i],hasLetterTypes)//code 3.7
            if hasNeededLetters(checkList[i],letterFreqs[i],hasLetters,blanks)://code 3.8
                res.append(word)
        i++
    return res
```

Code 3.5: countFrequencyOfEachLetterType
Used to count the frequency (how many) of each letter type in the string, and returns the result as a string of byte where index 0 contains the frequency of a, and index 0 b etc.
It has a linear time complexity depending on the length of the string used as input.

```
def byte[] countFrequencyOfEachLetterType(String string):
    #create byte array with size of alphabet
    freq=new byte['z'-'a'+1]
    #calculate the letter frequencies
    for each letter in string:
        #letter and 'a' are seen as ascii numbers
        #change so that a gives letterIndex 0 b gives 1 etc.
        letterIndex=letter-'a'
        freq[letterIndex]++
    return freq
```

Code 3.6: letterTypesInWord
Used to embed the information about which letter types are in the string into an integer. This way of saving the information makes it very fast to later check if a string has all needed letter types (see code 3.7). The time complexity of letterTypesInWord is linearly dependant on the length of the string.

```
#letterIndex 0 corresponds to a
#letterIndex 1 corresponds to b
#etc.

def int letterTypesInWord(String string)
    int res=0
    for each letter in string:
        letterIndex=letter-'a'
        # | is bitwise or,<< is bitwise shift left
        res=res | (1<<letterIndex)
    return res
```

Code 3.7: hasLetterTypes
This method is used to check if an integer (see code 3.6) has all the needed letters. This method is very fast and has constant time complexity.

```
boolean hasLetterTypes(int neededLetterTypes, int hasLetterTypes)
    return (neededLetterTypes & hasLetterTypes) == neededLetterTypes
```

Code 3.8: hasNeededLetters
This method is used to check if hasFreq and blanks has all the needed letter frequencies. This check has linear time complexity depending on the number of letter types in the string used to make the needFreq array (this is the same as the length of the checkList array). The maximum number of letter types are the same as the length of the alphabet.

```
boolean hasNeededLetters(byte[] checkList,byte[] needFreq, byte[] hasFreq,int blanks){
    for i in checkList:
        if needFreq[i]>hasFreq[i]:
            need=needFreq[i]-hasFreq[i]
            blanks=blanks-need
            #check if used too many blanks
            if blanks<0
                return false
    return true
```

3.7 Evaluation strategy

We intend to let our greedy bot play against a bot that plays the game by placing moves randomly. We do this to simulate how a average user plays the game versus our greedy bot, once a person finds a suitable move they are to lazy to find a better move or just can't find any better moves.
Once the bots has played several games with each other we should be able to see a pattern illustrating how the difference in strategies affects the amount of points they generate.
By doing this we will be able to draw a conclusion about our greedy bot, if it is a viable strategy or not.

We also intend to let out greedy bot play against itself to see how the amount of points are affected. Our hypothesis is that the average points will be lower because the other greedy bot "steals" good positions and uses more tiles so that there are less chances to get points.

4. Results

4.1 Evaluation

4.1.2 greedy versus random

The results from the games played was as expected, a major victory for the greedy bot against the random bot. As seen in figure 4.1 the greedy bot always has a large gap in the amount of points generated during the game and thus always ends up as the victor.

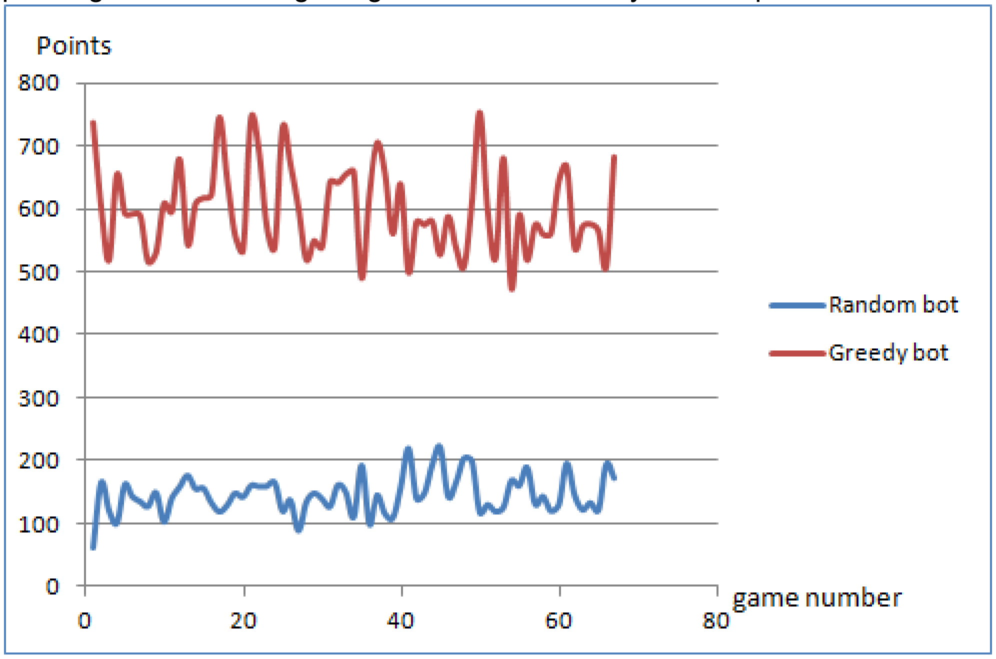


Figure 4.1 Points at the end of the game for 67 games played for greedy versus random bot

	Greedy	Random
Average	598,806	146,4776
Max	752	222
Min	475	63

Figure 4.2 Information for the 67 games played

4.1.3 Greedy versus Greedy

When playing the greedy versus greedy we see that the average amount of points generated drops due to the opponent playing smarter thus making it harder to score large amount of points. The opponents steals good placement positions and creates longer words leaving less tiles in the cache.

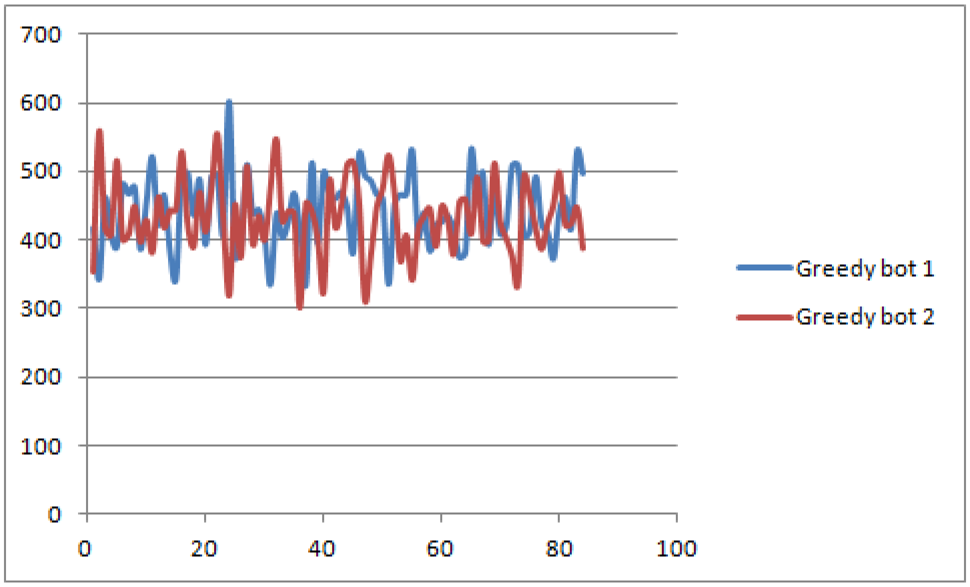


Figure 4.3 showing a greedy bot versus a greedy bot

	Greedy 1	Greedy 2
Averege	442,0476	431,8095
Max	602	558
Min	334	302

Figure 4.4 Additional information about greedy versus greedy

4.1.4 Time

As we can see in Figure 4.3 the time it takes for the bot to make a move is within an acceptable range. You may notice the big spikes in the graph, these occur when the bot receives a wildcard.
The average time it takes the bot to make a move is calculated as 39.7 milliseconds.
The time it takes for the bot to handle all communications with the server is excluded from this graph. This graph only has the actual time it takes for the bot to decide on what word it wants to play as this is the only thing we are interested in.

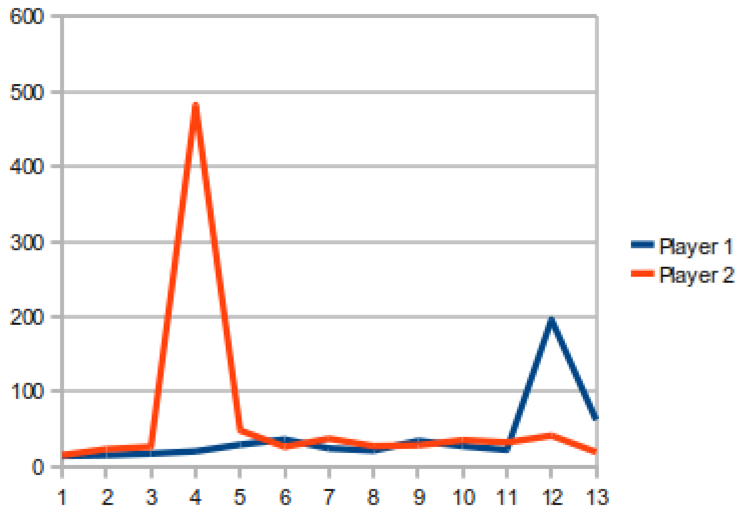


Figure 4.3 Example of time taken per move during a game in milliseconds with two greedy bots playing against each other

5. Discussion

5.1 Error Sources

5.1.1 Time

The time it takes to make the moves may be very different against a different opponent, because the time is dependant on how the board looks and how many possible moves there are, and that is dependant on what words are placed on the board.

The time is dependant on “random” factors such as if the computer is doing other things at the same time. This can be mitigated by taking the average of many runs.

The time is very dependant on the hardware and it will take less time on faster hardware.

5.1.2 Score

As can be seen in figure 4.1 the points of a game can vary. We mitigate this error source by making many runs and also displaying the data in such a way that these variations can be seen.

The greedy bot might not be as effective against a more difficult opponent. A more difficult opponent might block high scoring moves, or try to exploit the fact that the greedy bot is always trying to make the move with the most points.

5.2 Further work

5.2.1 advanced bot

We want to make an advanced bot but we will not make it due to time limitation. We want the advanced bot to consider the opponent's possible moves after a move, so that the points gained relative to the opponent can be maximized.

We could use the expectimax algorithm [12] to consider the weighted average (the weight is the probability of getting a rack) of every counter move to every possible move. This would also take into account what possible racks the opponent can have. Getting a rack is the random event and every possible outcome has to be considered, and there are 3222188 possible rack combinations at the start of the game. A lower estimate of the time would be 1 ms * 100 moves * 3222188 combinations = 89.5 hours. This amount of time is unacceptable and thus not possible for us to use this idea. But if the number of considered racks could be lowered in some way it might still be possible, or to use expectimax at the end of the game when the amount of possible racks are lowered due to the low amount of tiles in the cache.

It is probably more efficient and almost as accurate to do a Monte Carlo simulation[10]. This could be done by first finding all possible moves and then for each move: Do simulations with random racks and let greedy bot find the move with the most points for each random rack. Then calculate the average of these results and subtract it from the points of the move. The resulting points should be close to the result of using the expectimax algorithm, if enough simulations are run. If the simulations are fast enough we could even do the simulation a few steps into the future.

5.3 Conclusions

We can conclude from the results of the testing that the greedy bot does indeed work as a viable strategy for playing the Wordfeud game.

If we compare the average points for greedy vs random and greedy vs greedy (figure 4.2 and figure 4.4) we can see that the average points are lower against the greedy bot. This seems to support our hypothesis that the greedy bot will get less points against a more difficult opponent.

From what we read in the paper about MAVEN [5] we can conclude that our but could not play at championship level.

6. Reference List:

1. The wordfeud website containing most rules for the game.
   http://wordfeud.com/wf/help/ (11/4/2012)
2. The wordfeud application page on the android store.
   https://play.google.com/store/apps/details?id=com.hbwares.wordfeud.free (11/4/2012)
3. The scrabble board game.
   http://en.wikipedia.org/wiki/Scrabble (11/4/2012)
4. The DAWG data structure.
   http://en.wikipedia.org/wiki/Directed_acyclic_word_graph (11/4/2012)
5. Text about the Scrabble playing bot MAVEN Title: World-championship-caliber Scrabble
   Author: Brian Sheppard
   http://www.sciencedirect.com/science/article/pii/S0004370201001667 (11/4/2012)
6. The HTTP protocol.
   http://sv.wikipedia.org/wiki/HTTP (11/4/2012)
7. Python wordfeud client for PC.
   https://github.com/jonte/JarJar9 (11/4/2012)
8. The JSON data string format.
   http://www.json.org/ (11/4/2012)
9. Mastermind android application on android store.
   https://play.google.com/store/apps/details?id=se.ballefjongberga.wfmm&hl=en (12/4/
   2012)
10. Monte Carlo algorithm
    http://en.wikipedia.org/wiki/Monte_Carlo_algorithm (12/4/2012)
11. B* algorithm
    http://en.wikipedia.org/wiki/B* (12/4/2012)
12. Expectimax algorithm
    http://en.wikipedia.org/wiki/Expectimax (12/4/2012)