Greedy automatic Wordfeud bot

Vassili Arian
`arianv@kth.se`

Al-Qaysi Ibrahim
`iaq@kth.se`

April 4th, 2016

1 Background and purpose

The purpose of this project is to create a bot that plays the turn-based puzzle online multiplayer game called Wordfeud. The bot solves all valid moves available and automatically plays the best move based on a greedy algorithm. Because the game is turn-based and we use a greedy algorithm we can essentially eliminate external factors and implement the algorithm using parallelism without the threads ever needing to affect each other. To fully maximize the potential of a really fast parallelized bot, we also want to minimize the time complexity of the algorithms as much as possible.

2 The game

Wordfeud is a crossword turn-based puzzle game for the iOS, Android and Windows phones. The game is a basically a mobile multiplayer implementation of the classic board game known as "Scrabble". The rules are bent a little bit, but they are essentially the same and shares the same goal that is to form words that generates points. The winner is the player with the highest score at the end of the game.

2.1 Tiles

Just like in Scrabble, the letters used to form words are called tiles. Each tile has a predetermined score value, and is the basis of how much a word will be worth in points based on the coordinate of the tile on the game board. There exists 102 character tiles, and 2 blank tiles in the game. A blank tile can be assigned to any letter but doesn't generate any points.

2.2 Rack

Each player has a rack containing 7 random tiles. A player uses these 7 tiles, and the tiles already on the board to form words. The tiles from the rack that gets used during a move is replaced by new tiles so a player generally always has 7 tiles to work with. There is however a limit in how many tiles the game replaces, because there exists 104 tiles in total and 14 of these tiles (7 tiles to each player) gets distributed at the beginning, only 90 tiles in total can be replaced. A player can thus have less than 7 tiles in the rack at the final stages of the game.

2.3 Game board

The game board consists of a 15x15 grid where tiles can be placed. The first move of a game has to be placed with a tile at the center (7, 7) of the game board. The default normal game board has bonus tiles spread out in a pattern on the board (See Figure 1). These bonus tiles will further amplify a moves total score based on the tile that is placed on the bonus tile. A bonus tile can only be used once per game, but can affect multiple words on one move. They are also multiplicative, so multiple bonus tiles can affect each other during one move.

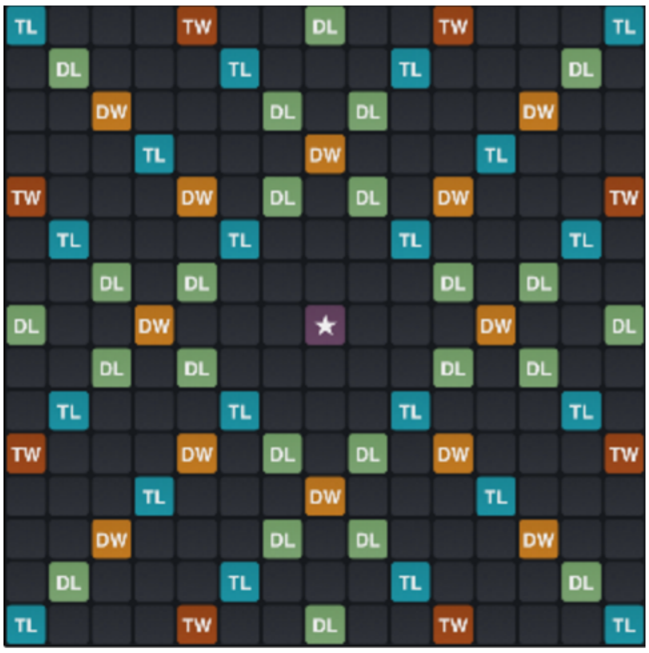

Figure 1: A normal game board

2.3.1 Bonus tiles

There exists 4 different types of bonus tiles:

- DL (Double Letter): The tiles character score that is placed on this bonus tile is multiplied by two.

- TL (Tripple Letter): The tiles character score that is placed on this bonus tile is multiplied by three.
- DW (Double Word): The total points for the words that crosses this bonus tile is multiplied by two.
- TW (Triple Word): The total points for the words that crosses this bonus tile is multiplied by three.

2.4 Move

A move is defined as the tiles that goes from the rack to the board. A move must consist of only one main word, which means that every tile in the move has to share either the same column or the same row. The move consists thus of the main word, and all the side-effect words that may be formed by the move. All words that are formed, including the side-effect words, must be valid words in the dictionary used by Wordfeud or the move is invalid. The words formed are defined as a connected chain of horizontally (left to right) or vertically (top to bottom) placed tiles on the board. Furthermore the move has to connect to at least one already placed tile to be considered valid (except on the first move).

2.5 Points

All words formed by a move generates points as defined in section 2.5.1. Each tile has a predetermined character value as described and exemplified in section 2.5.3, and together with bonus tiles they define the moves total score. If a player uses the entire rack (7 tiles) in a move, the move gets an extra +40 bonus points.

2.5.1 Score calculation of side-effect words

As previously described, a move may form multiple words in multiple dimensions. Each tile in the move may form a so called side-effect word in the opposite dimension of the main word together with already placed tiles on the board. All these side-effect words generates points and together defines the value of the move. A move with multiple side-effect words has a high correlation with the strongest move one can make.

2.5.2 An example of score calculation of a move

An example of a move with multiple side-effect words is provided in Figure 2. As we can see, the yellow tiles represents the move thus the main word of the move is the vertical "QI". If we iterate each tile of the move, the "Q" forms the horizontal side-effect word "QI" and the "I" forms the horizontal side-effect word "IN". To calculate the total score of the move, we start with calculating the score of main word then add the scores of the possible side-effect words. The main word consists of the tile ”Q”, which has 10 as character score, and the tile ”I” which has 1 as character score. Together the main word is worth 11 points. The ”I” tile of the move however, resides on the bonus tile ”DW” which means the final score of the main word is 22 points. Moving on to the side-effect words, as previously calculated the ”QI” side-effect word is worth 11 points. Because the ”I” in the side-effect word is not a tile in our move, the ”DL” bonus tile is already consumed thus will not multiply the ”I” tiles value. Finally the side-effect word ”IN” is worth 2 points, but because the ”I” which resides on the ”DW” bonus tile, is a tile in our move we can use this bonus tile again which brings the score of this side-effect word to 4 points. The final score of this move is thus 22+11+4=37 points.

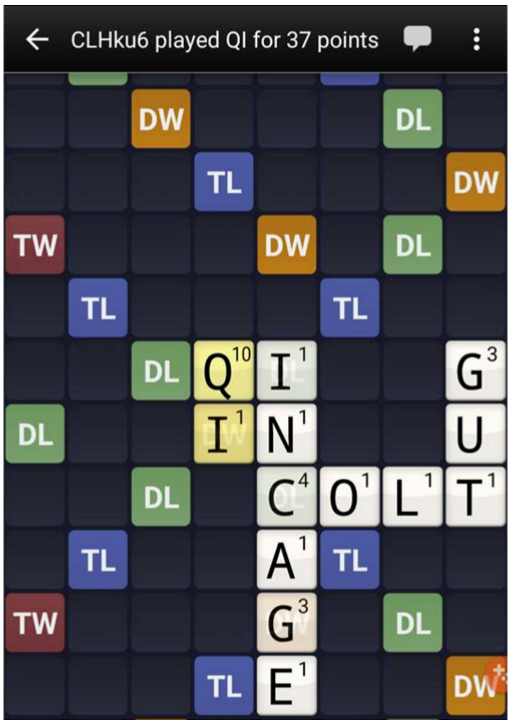

Figure 2: An example of a move with side-effect words

2.5.3 Predetermined character values

Wordfeud provides the character values of every alphabet it uses, and an example is included in this report as can be seen in Table 1. Important to note is that the bot also fully handles the Swedish alphabet and its dictionary. Theoretically the bot can handle every language Wordfeud can, but that is not something we have prioritized, thus is the bot currently limited to only English and Swedish games.

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

Table 1: English alphabet predetermined character values

3 The bot

3.1 Programming language and environment

This project is made solely in Java, and was written in NetBeans IDE. We also used GitHub as our version control system.

3.2 Pretending to be the app

First and foremost, to succeed with creating a bot that can automatically do everything from fetching game information to playing the optimal word, we need the bot to pretend to be the app. By sniffing the packets from the Wordfeud android app we could observe and re-implement their API in java. Fortunately the Wordfeud servers API was really easy to reverse-engineer due to their easy readable API URLs, and gave very readable JSON responses. For example, to retrieve account status information in JSON-format, we only needed to simply call the URL `api.wordfeud.com/wf/user/status/` with our session-id that was provided after authentication. By further analyzing the packets from the android app we managed to fetch info from specific games such as the rack and the tiles in the board.

3.3 Bot AI algorithm

The only viable approach that also fits this course curriculum is for the bot to use a greedy algorithm. This basically means that the bot does not look further ahead than what it can see on the board and in its rack. The bot finds all valid moves, and chooses the move with the highest score. This can be a huge limitation though since the bot does not take in consideration what possibilities that might open up for the opponent. However, we do get rid of all possible external factors and can instead create a very fast parallelized bot. The bot will probably lose to more advanced bots and advanced players but will probably crush the average player.

3.4 Solver approach and efficiency

We found another greedy wordfeud bot[1], but we did not like its approach at all. They filter their dictionary on every coordinate to only include all possible words that can be formed with the tiles in the rack and the tiles on the same row on the board. Then they try to place every word with the optimal length in the filtered dictionary with the coordinate as stem on the board, and then see if that word would result in a valid move or not. This is in many ways very inefficient because of the many constant time filtering functions, which lead to our approach which instead limits each coordinate and uses different optimized algorithms depending on what tile is on a coordinate. By having a already placed tile as a stem, we can figure out all possible prefixes that may form a word with our rack and that also ends with the tile, and then using the full prefix and the remaining rack to find all possible postfixes thus full words. With this approach we never try to complete a sub-word without making sure it may form a valid word, thus eliminating the need of filtering our dictionaries. This also means that tiles on the board with a non empty tile left of it is already a part of the left tile's every full prefix, which means we can skip them. Then we calculate the moves score and at the same time validating that the move doesn't generate any invalid side-effect words. By creating this generic method to find all horizontal solutions based on a coordinate as a stem, we can also parallelize this generic function to really make things fast. After finding all horizontal solutions, we rotate the board counter clockwise and flip it vertically. Then we reuse our horizontal function and thus will find all vertical solutions.

3.5 Dictionary datastructure

From a simple analysis, the biggest time-hog is to iterate the dictionary which in many cases can be very huge. With our prefix approach in mind, the datastrucure we chose were the node based Trie datastructure. A Trie, also called a prefix tree, is a tree datastructure that for any one word looks like a linked list. An example of Trie datastructure with some words already added is provided in Figure 3. In this Trie the gray nodes are represented as the word-nodes, which means that this particular Trie from left to right contains the words ”ace”, ”aced”, ”aces”, ”acre”, ”acres” etc. A Trie has a very fast prefix search time, and algorithms to find the different prefixes with a rack and finding full words with a prefix and a rack is further optimized by only iterating the rack and checking if the rack character is a valid child from a node instead of iterating every child from a node and checking if the child's character representation exists in the rack. Thus will the length of the rack be the relevant factor of the time-complexity of the different search algorithms. It's also further optimized by temporarily removing the rack's character from the rack when going deeper in the recursive functions, which means smaller iterations thus faster time complexity after each recursive call in the Trie.

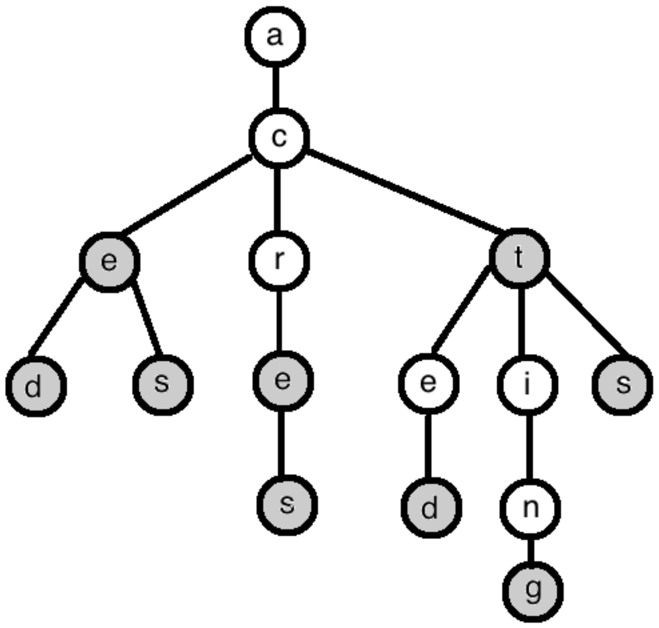

Figure 3: An example of a trie

3.6 Concurrency

The approach described in section 3.4 led us to create a generic function which found all valid horizontal solutions based on the row, the rack and the tiles on the board. The method iterates the columns of the row, uses the coordinate to find the solutions and finally adds the solutions to a collection. The parallelization of this method is thus easy, we initiate a ExecutorService with a fixedThreadPool, we initiate a runnable for each row (15) with all necessary game data, convert the runnables to callables, add the callables to a collection of callables, then call the method invokeAll() on the thread pool with the callable collection as argument. The callables will find solutions and add them to the only critical sector we have, which is the solution collection. Fortunately we can simply use a Synchronized HashSet as the solution collection which handles the critical sector for us.

4 Discussion

The bot is really fast, and is cross validated with different Wordfeud cheat services to ensure that it indeed finds every possible solution (it does). By emanating from locations with already known facts of the current state of the game, combined with never trying to generate or search for sub-words that we know are invalid in its location wrapped together with a concurrent thinking and smart data-structures really leads up to impressive and fast software. We could probably further optimize the dictionary datastructure by using a directed acyclic word graph instead of the Trie, but we felt that the Trie was really flexible and easily modified to fit our needs.

References

[1] F. E. Martin Berntsson, *Automatic Wordfeud Playing Bot.* [Online]. Available: http://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand12/Group3Johan/report/berntsson_ericsson_report.pdf