Reykjavík University

Artificial Intelligence

Final Project - Scrabble AI

Authors:
Eysteinn Gunnlaugsson
Magnús Sigurðarson
Sindri Már Kaldal

Supervisors:
Stephan Schiffel
Kári Halldórsson

July 18, 2018

Contents

1 Introduction

In this project we decided to implement a Scrabble bot.

Scrabble is a word game in which two to four players score points by placing tiles, each bearing a single letter, onto a game board which is divided into a 15×15 grid of squares. The tiles must form words which, in crossword fashion, flow left to right in rows or downwards in columns. The words must be defined in a standard dictionary.

In Scrabble, each letter has a certain score which is supposed to mirror how difficult it is to create words with that particular letter. In the official Icelandic Scrabble, the letter score does not reflect the Icelandic language accurately. Consequently the Icelandic Scrabble Society (I.S.S.) has made an effort to recalculate the letter-score and how many times each letter appears in the letter bag. I.S.S released their findings and in our A.I. we used their recalculated letter score and their new letter-bag.

With that being said, in this report we will outline the process of our implementation of a Scrabble A.I. that we believe can beat most human players.

2 Motivation

When we were informed that we could choose a subject for our final project we weren't quite sure what subject would suit us the best. Artificial intelligence can be used in numerous applications and the possibilities for us to choose from were endless. But we decided from the start that we would pick a project that we would be proud of sharing. Since the team consists of students that are starting to learn the field of artificial intelligence, it was integral to us that we would choose a subject that we could finish, but still keep it challenging, so we could take some knowledge and experience from the project.

When the idea of a Scrabble Bot came up, the team was a little skeptical. First of all, we didn't have a clear idea or a vision of how to implement it. Secondly, the algorithms and techniques that have been taught in the class weren't exactly tailor-made for this problem, so we were worried that we would not implement 'artificial intelligence', but rather just a basic dictionary look-up.

But the idea however intrigued us. Thus we did a bit of a research and stumbled upon a research paper from 1988 called *'The World's Fastest Scrabble Program'*[1].

As quoted from this research paper : "*An efficient backtracking algorithm makes possible a very fast program to play the SCRABBLE Brand Crossword Game. The efficiency is achieved by creating data structures before the backtracking search begins that serve both to focus the search and to make each step of the search fast.*"

We talked to our teachers, Stephan and Kári, and introduced to them this idea. After receiving positive feedback we decided to try and implement 'The World's Fastest Scrabble Program'.

[1]The World's Fastest Scrabble Program - Andrew W. Appel & Guy J. Jacobson

3 The Approach

As mentioned before, our number one goal was to hand in a project that was 'functional', i.e. not a half-finished program. Thus we set ourselves goals to get a minimum viable prototype as soon as possible.

To test the functionality of the AI and to be able to compete against it, we felt it was important that we had a GUI to show the state of the game. Since the GUI wasn't part of the A.I., we didn't want to spend too much time on implementing it and thus 'borrowed' the GUI from a GitHub repository[2] we found. That GUI implementation was easy to communicate with but lacked some information we wanted to include, so we modified it to contain the rack of player one, and the move history of both players, where the word and the score for that word is displayed.

Coincidentally, in that repository, a Scrabble bot had also been implemented. From that repository we took some ideas the we liked, such as implementing the *Move* class and other things.

To properly test the GUI we started by implementing the code to make it possible for human players to lay down tiles on the board. We did this the easy way to save time. The human player is simply asked for the x and y coordinates of where his word should start. Then he is asked whether the alignment should be horizontal and vertical and finally he is asked for the word he wants to lay down. Since this part of the Scrabble game was not part of artificial intelligence, we spent no effort to restrict the player to lay tiles from his rack or make sure that the moves are legal. Thus the human player can lay down any word he wishes down on the board.

When we successfully made the GUI show us the tiles laid down by the human player the real fun started, implementing the Scrabble bot. For future references in this report, let's call the Scrabble A.I. *AgentFresco*, since that is the name of the class that implements the logic.

The first problem we came across was how to search for possible words that AgentFresco could lay down. Our word collection consists of roughly 660.000 words and thus it was important that we could search through that data in a fast and an efficient way. We started by putting our word collection into a hash map, but that proved to be very inefficient. However, the research paper that we were basing our Scrabble bot on suggested structuring the lexicon as a $D.A.W.G$(Directed Acyclic

[2] `https://github.com/dianagr/Scrabble`

Word-Graph), which is a data-structure that stores the words in a finite state recognizer (see Figure 1) and speeds the search time immensely.

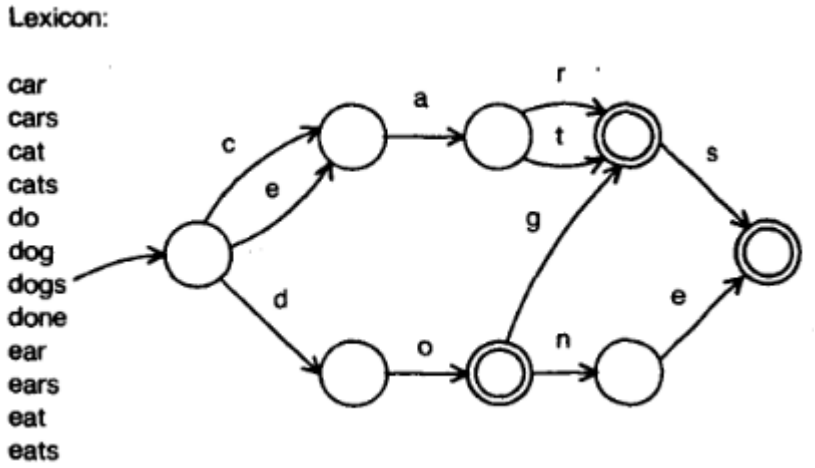

Figure 1: A D.A.W.G.

The search time is still linear to the length of the word, but that is good enough since players have only 7 tiles in their rack and the words are at most 15 characters. We used a implementation of the D.A.W.G. algorithm we found online[3]) and applied it on our lexicon. This implementation of the dawg came with some useful functions, such as *contains(String str)*, which checked if the string given as a parameter was a legal word and *getStringStartingWith(String prefix)*, which would return an *Iterable<String>* and was easy to check whether was empty or not. This would allow us to walk through the dawg easily.

Finding an implementation of the D.A.W.G. was really useful since that meant we had more time for implementing the algorithm outlined in the research paper *The World's Fastest Scrabble Program*. The algorithm isn't very straightforward and it took some time to wrap our heads around i t. An Icelandic website called `www.netskrafl.is` hosts an online Scrabble game where you can compete against Scrabble A.I. The GitHub repository[4] that hosts the code and is open according to the terms of the GNU General Public License v3, contains a good explanation of the code. We used the word collection from the same repository in our final version as it did not contain as many strange words as our initial lexicon.

The following explanation is taken from the above mentioned repository :

[3] `https://github.com/Qualtagh/DAWG`
[4] `https://github.com/vthorsteinsson/Netskrafl`

Moves are found by examining each one-dimensional Axis of the board in turn, i.e. 15 rows and 15 columns for a total of 30 axes. For each Axis an array of Squares is constructed. The cross-check set of each empty Square is calculated, i.e. the set of letters that form valid words by connecting with word parts across the square's Axis. To save processing time, the cross-check sets are also intersected with the letters in the rack, unless the rack contains a blank tile.

Any empty square with a non-null cross-check set or adjacent to a covered square within the axis is a potential anchor square. Each anchor square is examined in turn, from "left" to "right". The algorithm roughly proceeds as follows:

1. *Count the number of empty non-anchor squares to the left of the anchor. Call the number 'maxleft'.*

2. *Generate all permutations of rack tiles found by navigating from the root of the DAWG, of length 1..maxleft, i.e. all possible word beginnings from the rack.*

3. *For each such permutation, attempt to complete the word by placing the rest of the available tiles on the anchor square and to its right.*

4. *In any case, even if maxleft=0, place a starting tile on the anchor square and attempt to complete a word to its right.*

5. *When placing a tile on the anchor square or to its right, do so under three constraints: (a) the cross-check set of the square in question; (b) that there is a path in the DAWG corresponding to the tiles that have been laid down so far, incl. step 2 and 3; (c) a matching tile is still available in the rack (with blank tiles always matching).*

6. *If extending to the right and coming to a tile that is already on the board, it must correspond to the DAWG path being followed.*

7. *If we are running off the edge of the axis, or have come to an empty square, and we are at a final node in the DAWG indicating that a word is completed, we have a candidate move. Calculate its score and add it to the list of potential moves.*

After reading these well-explained instructions we had a clearer vision for how we wanted to proceed. We however did the algorithm a bit differently. Before finding a move we updated the anchors of the board. When the board is empty, the center square is the only anchor present on the board. Then, instead of examining the

board axis by axis, we searched through the whole board, and for each anchor we came across we searched for moves both horizontally and vertically. Since we didn't implement the axis method it is hard to tell which method is more efficient but this approach worked very well for us.

When looking at an anchor for moves, the crosscheck set of other empty squares must be updated, but for the opposite direction than is currently being examined for the anchor square. The crosscheck set of a square is the set of letters from the player's rack that can be legally laid down on that particular. Thus, if the empty square is next to a word currently on the board, the set will consist of letter that can be legally placed to extend that word.

With this information, the steps in the above instructions were implemented. Since the implementation of the D.A.W.G. we were using didn't use nodes and children, we had to work around it and use the function *getStringStartingWith(String prefix)*, as mentioned before. For each permutation AgentFresco generated as outlined in step 2, we tried adding letters that were both in our rack and in in the crosscheck set of the current square. If a word existed in our dictionary that started with permutation and the added letter, we moved to the next square and did the same thing. If we stumbled upon a legal word, we would save it if it was the current best move.

In an attempt to make AgentFresco even more 'intelligent', we wanted to do something similar to MiniMax and examine future moves that the opposite player would make. But to test what move the opposite player is going to make is difficult, since it's very difficult to predict what letters the opponent's rack contains until the very end. We thus decided to only look ahead if the bag of letters was empty, because then we we could know for certain what letters the opponent had by examining the letters on the board and in our agent's rack. When AgentFresco 'looks ahead', we generate all the possible legal moves for the current board. Then we examine the 10 moves that give the highest score and for each move, check what the best move the opponent can make. The best move is then chosen based one the very simple calculation :

bestMove.getScore() - opponentBestMove.getScore()

The move that yields the highest difference in favor of the current player is chosen as the best move. This adds a little 'nasty' element to our agent since it tries to minimize the opponent's score during the last few rounds.

4 Test cases and results

4.1 Lookahead

First we want to compare an AgentFresco with and without a one ply lookahead. We calculate the *difference* between our potential moves and the opponents best move. The idea was that in some cases our best move would not necessarily be the word that gives us the most points, but the word that sabotages *his* best move. The following output illustrates the *difference* of a few of our possible moves. The first move is the move we would have chosen without using the lookahead.

```
Current difference : 9
Current difference : 9
Current difference : 7
Current difference : 12
Current difference : 12
Current difference : 11
Current difference : 11
Current difference : 10
Current difference : 10
Current difference : 10
Word Chosen : ÆRÐUR with difference of 12
```

As we can see our lookahead did affect our decision, i.e. we were able to sabotage our opponents best move. Because we reevaluated our moves according to his move we see that choice number 4 gives us the best difference, and is the move we choose to play. This proves our hypothesis that using the word with the highest score is not necessarily the best move if you know the current rack of your opponent. We had to run the program around 5 times to get into a position like this one to see the lookahead have any affect on the game.

4.2 Score

Needless to say, none of us (or our friends) were able to defeat the final version of AgentFresco so we will not include the results of those matches in this report. The following table shows the score two instances of AgentFresco got when playing against each other, the first player using the lookahead explained before and the other not.

Match nr	Player 1 score	Player 2 score
1	354	317
2	382	336
3	324	322
4	351	322
5	332	329
6	404	317
7	370	350
8	343	284
9	421	361
10	346	412

The average score in these 10 matches for player 1 was: 362,7 and the standard deviation was: 31,35. Player 2 had an average score of 335, and the standard deviation was: 34,08. The players took turns of being the first one to lay down a word in each new game. These numbers show that the bot using lookahead is stronger and wins 9/10 games and has a significantly higher average score.
Comparing these scores to the scores participants in the Icelandic National Scrabble Competition (I.N.S.C) got, we assume that AgentFresco would do pretty well. The highest score at I.N.S.C. was 646 which is almost double our average score, so our agent rarely reaches that high scores. But as we know by now, Scrabble is not a game of highscores, but consistency and considering that our standard deviation is only 31,35, he is pretty consistent.

5 Conclusion

Looking back at this project, we are very pleased with the decision we made to implement a Scrabble bot. This was an interesting take on artificial intelligence and showed us yet another application where it can be used in an efficient and practical way. However, with the partially-known state space it was difficult for us to properly test the efficiency of the players. The score swung up and down depending the letters the were given randomly from the bag and thus for example testing the look-ahead method was made harder since in many cases he lost the last rounds due to being dealt worse letters.

We we're however pleased with the results we had. The agent never lays down an illegal move and often lays down clever moves that we ourselves would have never been able to spot. In our minds it would be capable of beating most human players, although he would most likely struggle against professional Scrabble players.

5.1 Potential future work

A feature that we would have loved to add is to predict the letters that the opponent had not just in the last stages of the game. If our agent possessed that information throughout the game, the agent could always choose the move that would give him the best score compared to the move the opponent would make. That would greatly improve the odds of the bot winning the game.

In the early stages of the game, it is very hard to predict the letters of the opponent and thus we were afraid with the limited time given, that if we would wrongly guess the letters, that it would perhaps yield worse moves for our agent than he otherwise would make.

6 References and existing work

Through the years, many different versions of a Scrabble A.I. have been implemented. As mentioned earlier in this report we got our idea from the breakthrough research paper *The World's Fastest Scrabble Program* which has inspired many other versions since it's release. The website `www.netskrafl.is` was a big help to us, especially regarding the explanation the can be read in the previous chapter.

The following list contains links to all the material we used during this project.

- The World's Fastest Scrabble Program
- The repository for www.netskrafl.is
- The repository that contained the Scrabble GUI
- The repository for the D.A.W.G.