Scrabble AI

Murat Şahin
TOBB ETU - YAP 441
191101010

***Abstract*—This project presents a Scrabble game and an AI agent that can play the game against a human opponent. The AI agent uses several techniques to find all possible moves, calculate their scores, simulate the play and score the simulations. The AI agent also supports multiple languages, such as English and Turkish. The results of the implementation demonstrate the feasibility of creating a functional and enjoyable Scrabble game with an AI component.**

I. Introduction

Scrabble is a popular word game that involves forming words from a set of letter tiles and placing them on a board. The game requires both linguistic and strategic skills, as players have to find the best words and positions to maximize their scores. Scrabble is also a challenging domain for artificial intelligence (AI), as it involves many subfields.

In this project, I implemented a Scrabble game and an artificial intelligence (AI) agent that can play the game against a human opponent. The AI agent uses several techniques to find all possible moves, calculate their scores, simulate the play and score the simulations. The AI agent also supports multiple languages, such as English and Turkish, by using different dictionaries and letter distributions. The implementation uses various data structures and algorithms to efficiently perform these tasks. The game interface is user-friendly and allows players to easily play against the AI.

The rest of this report is organized as follows: Section 2 describes the rules and features of Scrabble. Section 3 explains the design and implementation of the Scrabble game. Section 4 explains the design and implementation of the AI agent. Section 5 presents the experimental results and analysis. Section 6 discusses the limitations and future work. Section 7 concludes the report.

II. Scrabble

The basic objective of Scrabble is to play tiles marked with letters on a 15x15 grid to form words. After the initial word is played, players take turns adding words to existing letters.

If players are not satisfied with their tiles, they have the option to swap them by skipping their turn.

The letters placed in a single turn must form a single word from the dictionary, with no gaps, and must all be in either a horizontal row or a vertical column. Each new word must connect to the existing words in one of the following ways:

- Adding one or more letters to a word or letters already on the board.
- Placing a word perpendicular to a word already on the board. The new word must use one of the letters already on the board or must add a letter to an existing word.
- Placing a complete word parallel to a word already played so that adjacent letters also form complete words.

Any new words formed by these connections must also be in the dictionary.

Fig. 1. Scrabble board (real)

Each tile has a letter and a number on it. The letter is used to construct the words and the number indicates it's point value. Looking at the figure 1, the colors on the cells mark the premium squares and letters or words that played on those squares are worth more points.

To calculate the score of a move, players need to add up the point values of all the letters in the word, multiply them by any letter bonuses, then multiply them by any word bonuses. If a player uses all seven tiles from their rack in one move, they get an extra 50 points. If a player forms more than one word in a move, each word is scored separately.

At the end, the player with the highest score wins. The game ends when one of the followings happen:

- The pouch is empty and one player places his/her last tile.
- Both players exchange tiles twice in a row.

III. Game Implementation

I implemented the game in Python, specifically using PyGame library. Figure 2 shows the structure of the project. Each part is in its own module, namings are quite self-explanatory.

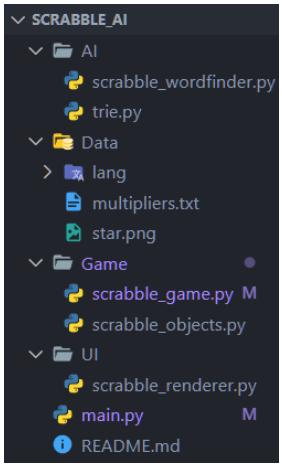

Fig. 2. Project structure

Implementation is very similar to its board game version, it includes basic functionalities like play and swap with some additions.

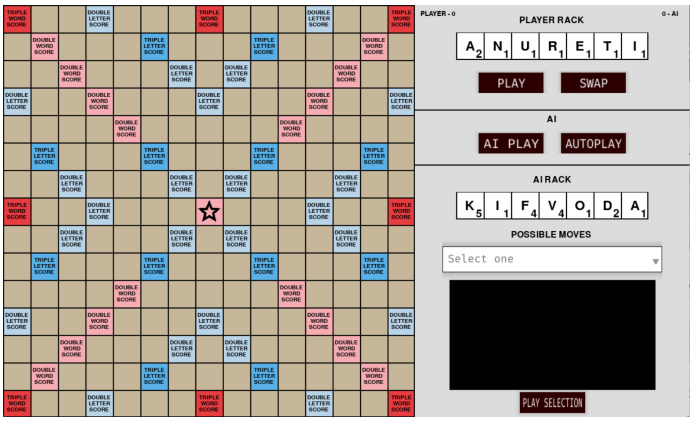

Fig. 3. Scrabble board (game)

Player can make its move by dragging and dropping tiles from his/her rack. Also, swapping tiles is possible by clicking swap and selecting tiles.

After that, if autoplay is selected, AI will make its move automatically. Else, AI Play button can be clicked for AI to make move. AI Play button can be clicked anytime, so that player can use it too.

When it is AI turn and autoplay is off, user can click dropdown menu to see all of the possible moves with its stats, which can be seen on figure 10.

All of the point calculations, move validation, random tile distributing is done.

IV. AI Implementation

For every turn, AI needs to find a play that consists of using tiles in its rack to form new word(s) on the board. In order to achieve that, method similar to described in [1] is used.

We can divide the process as follows: *the dictionary*, *anchors*, *forming words*, *move evaluation*, *move selection*.

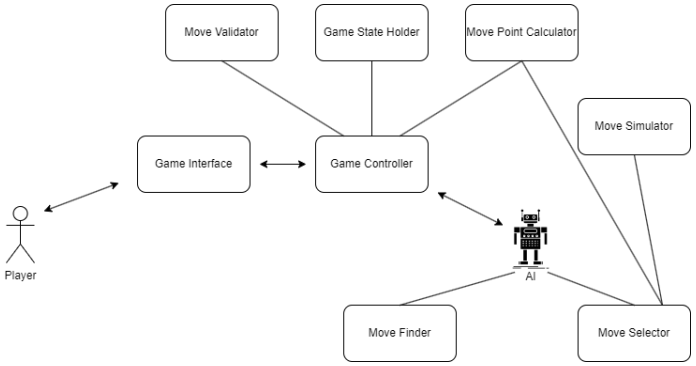

Fig. 4. Project workflow

A. The dictionary

First of all, we need to find a dictionary and a way to represent it in our program. I used several word lists I found as a dictionary, which I will talk more about in Section 5. *Trie* data structure is used to hold the dictionary in memory for program to use.

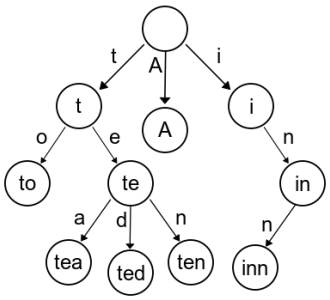

Fig. 5. Trie data structure

This data structure allows us to progress cumulatively while searching for new words and validating them. It is also way more efficient than holding a dictionary as a plain text list.

B. Anchors

In order for AI to find possible moves, it needs to know where to start searching for possible words. According to the Scrabble rules, players need to form words that is anchored to existing ones. Therefore, it is sensible to search from all of the empty neighbors of existing tiles, which we will call *anchors*. Figure 6 shows them.

C. Forming words

I will explain the algorithms a little superficially so as not to overwhelm you with details. After finding the anchor points, we now have baseline to conduct search. Consider a single anchor point and assume that we are searching words horizontally, our algorithm consists of following steps:

Fig. 6. Anchors numbered

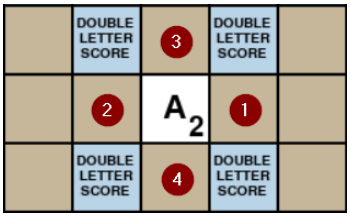

Fig. 7. Board state with AI Rack : D-S-E-A-O-A-I

1) Extending after the anchor: Assume that left of the anchor is not empty. So, there is a word already formed before. Get that words final trie node, we will extend from there.

a) Check if tiles in the players rack contains children of that trie node.
b) If it contains, that means that we can put that tile and continue extending the word.
c) If it doesn't contain, but the next cell on the board is filled with that letter, we can still continue extending the word.
d) For each step, check if that node is a valid word and continue until it reaches an end.
e) Apply these steps for each possibility using recursion

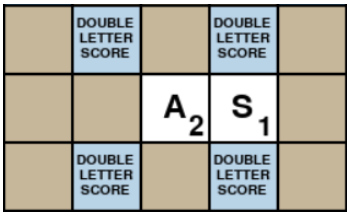

Fig. 8. Board state after right method on 1st anchor on fig 7

2) Find letters before anchor: If left of the anchor is empty, this means that we can put letters there and slide new letters until it goes out of bounds or collides with another anchor (in that case, extending right from that anchor already considers that case, no information lost). Then, we can start extending after the anchor with leftover player tiles.

a) Start from the root node for the first call
b) Check if tiles in the players rack contains children of that trie node.
c) If it contains, put that tile and slide already put tiles to the left.
d) Call extend_after function described above
e) Apply these steps for each possibility using recursion

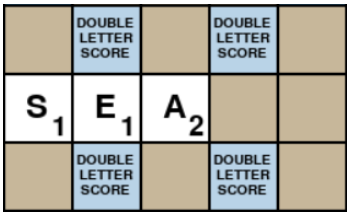

Fig. 9. Board State after left method on 2nd anchor on fig 7

Also, note that while placing the tiles, it may collide with another word and form an *invalid word* alongside our main valid word. This is solved by calculating the valid values for each cell on the board and checking it in every move. In short, those invalid moves are discarded.

I explained the horizontal search but vertical is the same with directions changed. Both of these directions are used in the code properly. Calling suitable functions for all of the anchor points returns every move that is possible to play on the board.

D. Move Evaluation

Now, we have all of the valid moves in our hands. We need a find a way to rank those moves, in order to choose which one to play.

For each move, we can simply calculate the formed word(s) points and possible total score gain. After that, the move with highest score gain can be chosen. This is the approach that worked quite well and used mainly in the project.

We can also think of a method similar to minimax algorithm, however it needs many simplifications since Scrabble is a game where players don't know each others current tiles and upcoming tiles. Lets define a basic simulation.

```
total_sim_score ← 0
for i in range(sim_amount) do
    sim_score ← 0
    for i in range(sim_depth) do
        Play the candidate move for the current player
        sim_score + = score_gain # can be negative
    end for
    total_sim_score + = sim_score
    Roll back changes
end for
return total_sim_score / sim_amount
```

This algorithm will give the average score gain for *sim_amount* times of simulation, with the depth of *sim_depth* moves.

E. Move Selection

I thought about using simulation results for move selection, since it takes future moves into account too. However, since I implemented the project in Python and it is quite slow for these kind of tasks, complexity that it brings is not worth in terms of performance and speed trade-off. Therefore, selecting moves in terms of instant score gain is the method I chose. Using simulation with a small depth and amount is a valid option too.

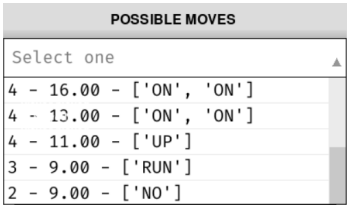

Fig. 10. Possible moves menu

Figure 10 shows the dropdown menu in the application that shows score gain, simulation score, words formed of the possible AI moves, with the corresponding order.

V. Experiments

This section contains examples from complete ended games. For demonstration purposes, best moves are played by human player too.

Fig. 11. Game result, 1000 words, English

Figure 11 shows the results using 1000 word English dictionary.

Figure 12 shows the result using 1000 word Turkish dictionary.

Figure 13 shows the result using 75000 word Turkish dictionary.

Note the difference in complexity of words formed between 1000 word and 75000 word dictionary.

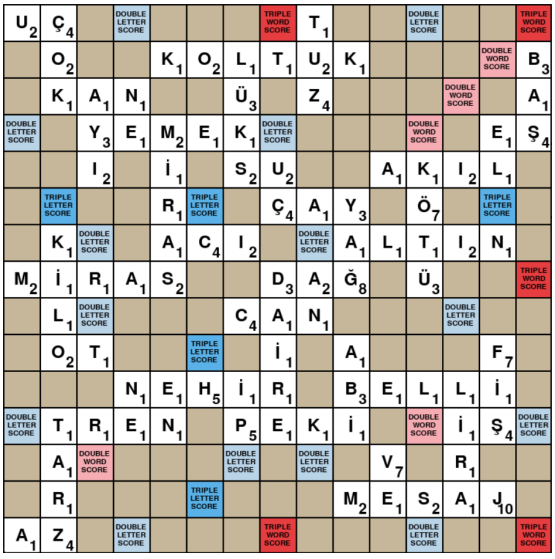

Fig. 12. Game result, 1000 words, Turkish

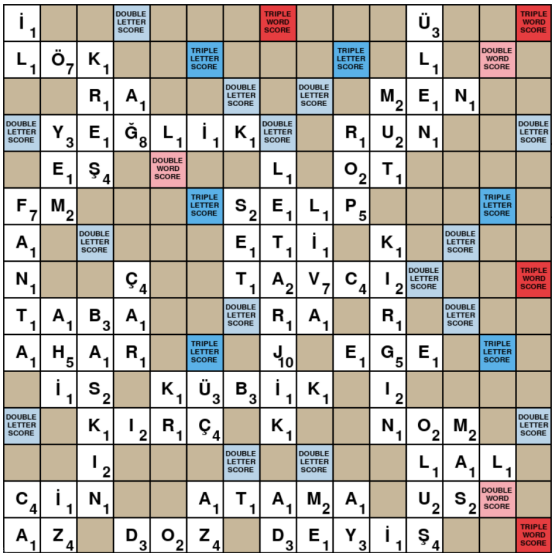

Fig. 13. Game result, 75000 words, Turkish

I was not able to measure the running times properly because it hugely varies on current game state and anchor amounts, not only dictionary size.

VI. Future work

There are several possible ways to improve the performance and efficiency of the AI agent.

- The current implementation uses Python, however it is not the most efficient language in terms of speed and memory usage. A more efficient language such as C++ or Java could be used to implement the game and the AI agent.
- Trie data structure is used to store the directory in the game. Directed acyclic word graph (DAWG) can reduce

the memory usage drastically compared to a trie, while still allowing fast word lookup and prefix search.

- A more adaptive heuristic function that changes according to the game state could be developed and tested. For example, at the beginning of the game, it may be more beneficial to play longer words and use more tiles, while at the end of the game, it may be more important to block the opponent's moves and avoid leaving open spaces.

VII. Conclusion

In this project, I was able to successfully create a functional Scrabble game and implement an AI agent that can play reasonably well against a human player. The AI agent was able to find high-scoring moves in most cases and challenge the human player with its moves. The source code of the project can be reached from the GitHub repository [2].

References

[1] Appel, Andrew W. and Guy J. Jacobson. "The world's fastest Scrabble program." Commun. ACM 31 (1988): 572-578.

[2] GitHub repository of the project. https://github.com/sta314/Scrabble_AI