Final Report: Solving Scrabble using AI Strategies

Daksh Narang, Gabriel Ronan, Justin Wang
University of Michigan
{dnarang, gronan, justwang} @umich.edu

1. Problem Description

1.1. Task Decription

Scrabble is a 2-4 player word-building board game. Words are formed using tiles containing letters, each letter worth different point values. The words formed should be from a standard dictionary. In addition, the words should be created in a way such that they can be read downwards in columns or left to right in rows, while diagonal words are not allowed. Furthermore, the word formed should have at least one tile placed next to an existing tile on the board. The 15x15 Scrabble board consists of some special tile locations, which if utilized while constructing the word, is worth more points. The game begins with 100 letter tiles in a letter bag. Each player draws 7 tiles out of a bag at first, followed by taking turns in building words with those letters. The player with the letter closest to 'A' starts first. As and when a player forms a word, they draw tiles out of the bag, and the number of tiles drawn equals the tiles used to form the word. The game ends when either a player is out of tiles or no more tiles are left in the bag and a player has used their last letter. The score of a player is the sum of scores across all turns and the player with the highest score wins.
We particularly aimed at reproducing and then improving on one of the existing algorithms, initially Q-learning, to enhance performance in the game by creating a learning-based agent. However, the existing code repository had a lot of missing code, even for the baseline algorithm. We decided to build on further from whatever we could extract from the repository. We pivoted from the existing algorithm to first try and implement the baseline code which is basically a greedy search algorithm that selects the word with the highest score. Then we tried to improve the agent's performance by adopting a probabilistic heuristic [6] for comparison. This heuristic is essentially a look-ahead search done for several plys of the game.

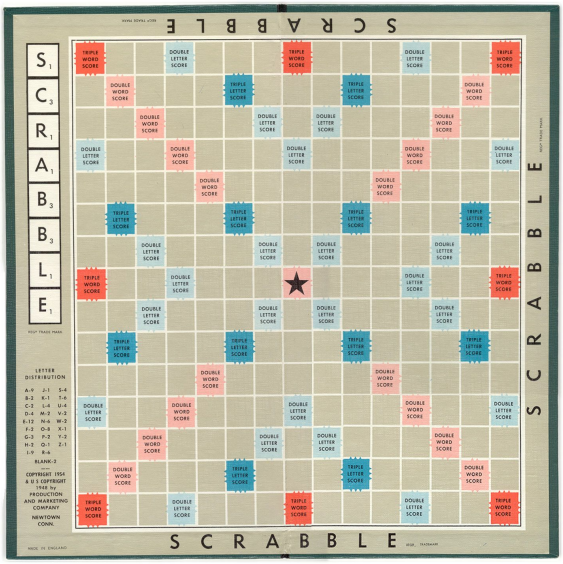

Figure 1. An empty Scrabble board [1]

1.2. Motivation

Scrabble is a good platform for testing artificial intelligence techniques. The game has an active competition scene with numerous national and international associations and a biennial world championship for English Language players. Compared to Scrabble, games like Go and Chess have small action spaces and configurations. In Scrabble, the number of possible moves is a permutation of both spaces on the board and words in a language, the total number of all possible actions is astronomical. Tackling these issues will enable the application of these algorithms in environments where actions take value in a continuum like in the realm of robotics or for more complicated tasks like self-driving cars. Although prior works show that computers can play better than humans, Scrabble is not a solved game. Furthermore, advanced computer Scrabble agents are of huge advantage to expert human Scrabble players because they rely on these programs

to improve their play by analyzing previous games and identifying where suboptimal decisions were made [7].

1.3. System Input

Since we have 2 models, the input to them is quite similar with just a few differences:

1. Common to both models: The board with Tiles, some of which have a multiplier assigned to them(special tiles, as described in the rules). Depending on the multiplier, if that tile position is used during word formation, it either doubles or triples the letter or word score.
2. Common to both models: 2 players with zero initial score.
3. Common to both models: A bag of letters denoting the frequency of each letter in the bag, and specific letter scores
4. Common to both models: Number of simulations to run
5. Probability Heuristic Model only: The number of plys to evaluate before making a move for the player

1.4. System Output

The output for both models is as follows:

1. The number of wins for each player
2. Average win score of the player
3. Final board state

1.5. Challenges Encountered

The game itself and its rules are easy to play and understand. A closer look reveals that implementation is rather challenging.

- The game has huge action spaces and configurations compared to games like Go and Chess. Moreover, Scrabble is not concerned with exhaustively searching the state space. Further, the letters are picked out randomly and the letter in the opponent player's racks are unknown, this random component gives the state space high variance. Hence, evaluation takes precedence over search because the outcome of the move cannot be calculated.
- The rules of the game govern that the new word must connect to an existing letter on the board. DAWG and GADDAG are 2 commonly used move generation algorithms in Scrabble. DAWG would perform many redundant searches before converging onto the solution for a move, the prefix storage ability of GADDAG helps overcome it.
- The paper we were trying to base our implementation on, had a lot of missing code in its repository. Further, we tried contacting the authors of the original paper but did not receive any response. Even the baseline algorithm was not working correctly. We first implemented the baseline model.
- Given our current knowledge, we first tried to implement Q-learning by ourselves. Given the huge action space and no defined state, it was hard to develop a Q-state matrix.
- We pivoted our development to a probabilistic heuristic for move evaluation. We all are really proud that we could contribute towards implementing two AI algorithms for the game and compare them.

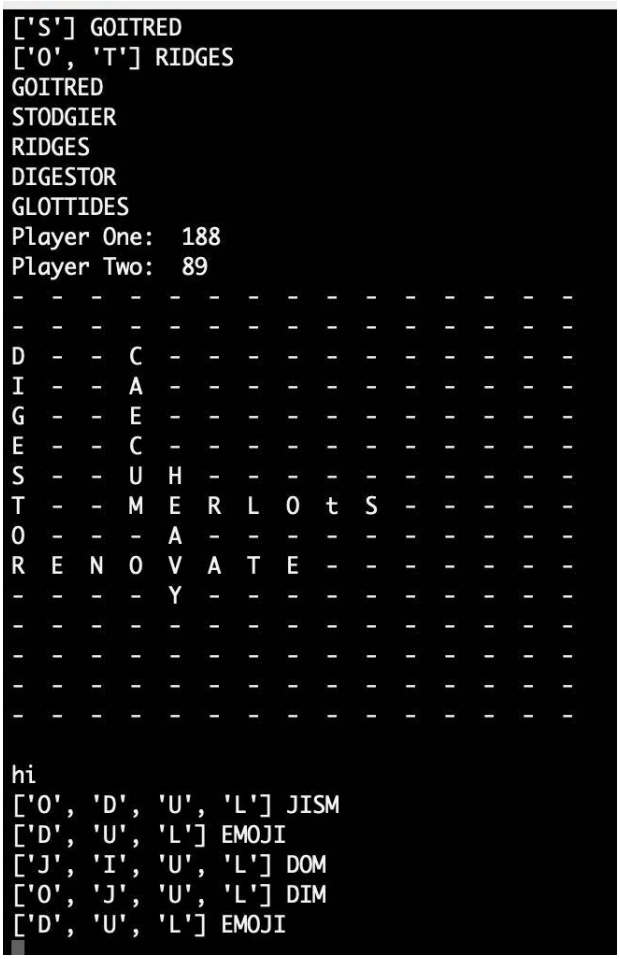

Figure 2. Board Representation

- The blank tiles in the game particularly slowed down the algorithm.
- The lookahead algorithm itself took a lot of time as we increased the depth for evaluation.

1.6. Contribution and Task Allocation

We tried to divide work equally amongst ourselves. Daksh researched existing methodologies, helped in fixing the baseline code, develop and debug the environment, contributed towards designing the probabilistic heuristic algorithm, ran experiments for comparison, and tried researching the Q-learning algorithm. Justin researched existing methodologies, fixed the baseline algorithm, develop and debug the environment, and contributed towards designing the probabilistic heuristic algorithm. Gabriel researched existing methodologies, helped fix the baseline algorithm, develop and debug the environment, designed and implemented the probabilistic heuristic algorithm, and ran experiments.

2. Related Work

2.1. Related and Recent Work in the Field

Initially, an efficient backtracking algorithm was proposed. It made possible a fast program to play Scrabble. The efficiency was 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 [2]. Maven is another AI Scrabble player that is claimed to be one of the best agents for solving the game and is now used in officially licensed Hasbro Scrabble games. The algorithms divide the gameplay into three phases: Mid-Game, Pre-Endgame, and Endgame. The 1 endgame starts when the last tile is drawn. Maven uses B*-search to tackle this phase and is supposedly nearly optimal. Little information is available about the tactics used in the pre-endgame phase, but the goal of that module is to achieve a favorable endgame situation [7]. Maven generally uses two- to four-ply searches in its simulations.
Quackle is another program among the best Scrabble agents. It includes a move generator, move evaluator, simulator, and user interface that can be used with any board layout, alphabet, lexicon, and tile distribution. Quackle uses a similar approach as Maven in the midgame phase. The heuristic function used is a sum of the move score and the value of the rack leave, which is computed favoring both the chance of playing a strong word on the next turn as well as leaves that garnered high scores when they appeared on a player's rack in a sample of Quackle vs Quackle games. The win percentage of each move is then estimated and the move with the highest win percentage is played. Researchers have also used data structures like DAWG and GADDAG for faster move generation [3].
Some works also quantify the value of knowing what letters the opponent has. It uses observations from previous plays to predict what tiles the opponent may hold and then use this information to guide the play. Opponent model is based on Bayes' theorem which sacrifices accuracy for simplicity and ease of computation [7]. Monte Carlo Tree Search (MCTS) methods have proven powerful in planning for large-scale sequential decision-making problems. MCTS methods build lookahead trees to estimate the action values, which is usually time-consuming and memory-demanding. Moreover, MCTS methods suffer when the planning depth and sampling trajectories are limited compared to the delay of rewards or when the rewards are sparse.

2.2. Future Work

Our model uses a probabilistic heuristic model which has not received a lot of attention. The model is relatively fast when compared to the run times of reinforcement learning techniques. Moreover, it performs fairly well against the baseline program which follows a greedy approach. Future work can be testing the model by increasing the depth for move evaluation before actually playing it. Also, we can tune the hyperparameters (defined in evaluation metrics) by increasing numAvg, k, and n, and optimizing bagNum using more powerful hardware to increase the performance of the probability heuristic model.

3. Methodology

3.1. Environment

We inherit the environment from the repository [5] and work on top of it. We tailored the code and made it suit our needs. The environment consists of the following components:

3.1.1 Board:

The Board class consists of 15 * 15 squares, each containing a multiplier. When placing a tile on each square, the score will be multiplied by the multiplier range from 1 to 3. The class also has methods to generate possible moves and calculate the current score. Another method checks whether the move is valid for all the intersecting points by maintaining the cross sets.

3.1.2 Player:

The player class is used to store the status of each player, including the current rack and score.

```
Gen(pos, word, rack, arc):                {pos = offset from anchor square}
   IF a letter, L, is already on this square THEN
      GoOn(pos, L, word, rack, NextArc(arc, L), arc)
   ELSE IF letters remain on the rack THEN
      FOR each letter on the rack, L, allowed on this square
         GoOn(pos, L, word, rack - L, NextArc(arc, L), arc)
      IF the rack contains a BLANK THEN
         FOR each letter the BLANK could be, L, allowed on this square
            GoOn(pos, L, word, rack - BLANK, NextArc(arc, L), arc)

GoOn(pos, L, word, rack, NewArc, OldArc):
   IF pos ≤ 0 THEN                                          {moving left:}
      word ← L ‖ word
      IF L on OldArc & no letter directly left THEN RecordPlay
      IF NewArc ≠ 0 THEN
         IF room to the left THEN Gen(pos-1, word, rack, NewArc)
         NewArc ← NextArc(NewArc, 0)                      {shift direction:}
         IF NewArc ≠ 0, no letter directly left & room to the right THEN
            Gen(1, word, rack, NewArc)
   ELSE IF pos > 0 THEN                                     {moving right:}
      word ← word ‖ L
      IF L on OldArc & no letter directly right THEN RecordPlay
      IF NewArc ≠ 0 & room to the right THEN
         Gen(pos+1, word, rack, NewArc)
```

Figure 3. Word generation with GADDAG

3.1.3 Tile:

The tile class is used to store letters that the game used.

3.1.4 Game:

The game object stores all the information for each game. It also defines the end-game condition and method to place a move.

3.1.5 GADDAG:

Published by Gorden et al. [4], the GADDAG is the data structure for the dictionary used for the game. It is a two-way directed acyclic graph where each node stores an array of letters. Each path in the graph is a valid word for the game. Different from the common data structure Trie (pronounced the same as the word "try"), each node can be pointed by multiple nodes, which largely decreases both time and space complexity. On the other hand, the two-way edge is especially useful for Scrabble since searching from the middle of the word is common.

3.1.6 Simulation:

In our simulation, we place two players against each other. Each player takes turns to make moves based on the strategy each agent uses. The board status and word choice can be monitored at every turn.

3.2. Algorithms

We developed the following algorithms for the agents to play Scrabble:

3.2.1 Greedy Search:

The most straightforward strategy one can think of is to choose the move with the highest score. Based on the current rack and board, the agent first generates all the possible moves one can play. We used the GADDAG move generation algorithm here(Figure 3). It then calculates the score for each move and sorts them based on the score. Finally, the agent simply chooses the first move in the array. The greedy search is used as our baseline algorithm.

3.2.2 Probabilistic Search:

Apart from only considering the current move, the probabilistic search algorithm performs a look-ahead and takes the opponent's moves into account. To implement the probabilistic search, the agent simulates the following k turns. However, since the agent is not supposed to know its opponent's rack, it generates several random racks. It then performs the greedy search on each generated rack and calculates the average score. If the k is greater or equal to one, there will be the agent's turn. In this case, it adds the average score played by the agent and subtracts the average score played by the opponent. With all the values, the agent can choose the move with the highest score difference estimated from the future k turns. The diagrams below show the decision tree structures for k = 1 and k = 2 probabilistic search. Nodes next to a summation sign (capital sigma) indicate that there are multiple random racks generated at that step. All other nodes use only a single random rack because they must make a single decision at that step.

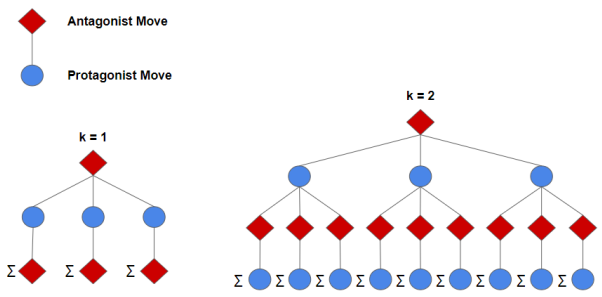

Figure 4. Decision tree structure of probabilistic search for k = 1 and k = 2 (n = 3 in both cases).

4. Experiments

4.1. Evaluation Metrics

None of our models required any training sets. We evaluated our models by playing the agents against each other in a head-to-head scenario. When playing the greedy agent against itself, we found that the

player that makes the first move wins ≈ 57% of the time. This is due to the double word score at the center of the board that the first player is required to use. We varied several hyperparameters of the probabilistic heuristic model and played each variation against the greedy model 100 times, allowing each agent to make the first move 50 times. The model that is able to obtain the most wins is therefore the strongest. This is the most direct way to analyze model performance because the goal is to maximize the probability of winning. Another possible way to evaluate performance is to analyze the final word score of each model. This is useful for finding how well the probabilistic model is able to minimize its opponent's score. Although a high word score is correlated to a win, more wins are not guaranteed due to a higher average word score. Another valuable metric that is not recorded in these runs is standard deviation. We will show that the win rate results are fairly close between the baseline model and the variations of the probabilistic model. Standard deviation would help us analyze significant differences between models and individual runs. This remains a goal for future work.

4.2. Model Performance

The baseline model is the greedy model. The hyperparameters used define the probabilistic models are defined using the following variables. bagNum: Use the greedy model until the letters in the bag < bagNum. k: k-ply lookahead. n: Branch over the top n best moves at each lookahead step according to word score. numAvg: Number of times to randomly draw from the bag to compute average score on last iteration. In the following table we show the number of wins and average score for each probabilistic model against the greedy baseline model. The instances where our

bagNum	100	100	50	50	50	50	25	25	0
k	1	2	1	2	1	1	1	2	-
n	3	3	3	3	3	5	3	3	-
numAvg	10	10	10	10	100	10	10	10	-
Lookahead Wins	45	49	51	44	43	55	48	50	47
Greedy Wins	55	51	49	56	57	45	52	50	53
Lookahead Score	362.75	372.1	368.27	368.61	370.19	376.83	375	372.93	367.29
Greedy Score	376.78	374.49	366.19	380.86	374.82	370.75	377.12	376.35	378.12

Figure 5. Results for several variations of hyperparameters.

probabilistic model beat the baseline model are highlighted in green, while the instances where it tied are highlighted in yellow. Our models had the most success when bagNum = 50. This means that the model used the greedy algorithm until there were only 50 letters left in the bag. This narrows down the possible racks that the opponent could have. It is possible that the probabilistic method is not accurate when there are too many possible moves for the opponent. On the other hand, when you wait too long to turn on the probabilistic model, it seems that there are not enough moves left to gain a significant advantage, therefore we see that the probabilistic model is roughly equal in strength to the baseline when bagNum = 25. We hypothesized that increasing numAvg (the number of random draws) would give us better performance, but this was not the case according to our results when we increased numAvg from 10 to 100. The win rate dropped considerably. Further testing is required to validate this relationship. On the other hand, the model performed best when we increased n (branching factor for potential best moves). This may be due to the fact that the agent now has more opportunities to play a move in which the opposing player will not be able to reach a multiplier square as easily. In conclusion, we were able to define a probabilistic model that outperformed our baseline and ran relatively quickly, but in general we found that the greedy model was able to beat most of the other probabilistic model variations. This points to the fact that the randomness associated using the board state and unknown rack to simulate the future is too significant for a probabilistic hueristic model to manage under many circumstances. Future work could involve increasing the hyperparameters numAvg, k, and n, and optimizing bagNum using more powerful hardware to increase the performance of the probability heuristic model.

References

[1] Empty scrabble board, https://i.pinimg.com/originals/8f/32/23/8f32231922200662689c27c1f0f04ca8.jpg. 1

[2] ANDREW W. APPEL and GUY J. JACOBSON. The world's fastest data structures scrabble program. Communicntions of the ACM, 31, 1988. 3

[3] Steven A. Gordon. A faster scrabble move generation algorithm. Software: Practice and Experience, 24(2):219–232, 1994. 3

[4] Steven A. Gordon. A faster scrabble move generation algorithm. Softw. Pract. Exper., 24(2):219–232, feb 1994. 4

[5] B. Jiang, M. Du, and S. Masling. Learning a reward function for scrabble using q-learning, 2019. 3

[6] Carl O'Connor. Developing an artificial intelligence to play the board game scrabble. 1

[7] Mark Richards and Eyal Amir. Opponent modeling in scrabble. IJCAI, 07. 2, 3