Challenging the Assumptions of Scrabble Static Evaluation

Ethan Mathieu
ethan.mathieu@yale.edu

Advisor: Professor Timothy Barron, PhD
timothy.barron@yale.edu

A Senior Thesis as a partial fulfillment of requirements for the Bachelor of Science in Computer Science

Department of Computer Science
Yale University
Dec 12, 2024

Abstract

In this thesis, I explore improvements to rule-based Scrabble heuristics. Scrabble is a fascinating game from a strategy perspective, featuring a degree of randomness that hems the effectiveness of deterministic move planning while still being a deeply skill based contest. As the field tends more toward machine learning models that fine-tune their strategy with play, I seek to analyze experimentally if improvements to rule-based heuristics, which use a fixed algorithm to determine which moves to play, can still push the boundaries of computational Scrabble. Specifically, I investigate if improvements can be made to the open source Quackle player, a robust Scrabble C++ engine. I contend that challenging certain predetermined notions that govern Quackle's current rules can lead to a more formidable AI player.

Acknowledgements

I would first like to acknowledge my advisor Professor Tim Barron for his assistance and guidance throughout the duration of the project. Professor Barron was incredibly involved, and I would not have considered exploring certain avenues without his encouragement to do so. Second, I would like to acknowledge the Department of Computer Science for providing an environment which fostered my growth as a scientist, mathematician, statistician and engineer. Finally, I would like to acknowledge my mother, father and sister for their relentless championing of my success.

Contents

1 Introduction

1.1 Why Study Scrabble?

Scrabble is a crossword board game that allows players to create word combinations using tiles on a gridded board. The tiles themselves have their own value and a word's move value is derived from the sum of its tiles. Certain squares have special multipliers, called **premiums**, that change the move value by a factor. After making a move, players draw new tiles from the bag to replenish their 7-tile **rack**. Rather than play a word, players can instead **exchange** certain tiles for new ones or do nothing, passing creating a word at that point. The game is complete when the bag of tiles is empty or a player resigns. The player with the most points, determined by the words they played on the particular squares, is the winner. Should there be a tie in point totals, the game is a draw.

Figure 1.1: A Scrabble Board

While Scrabble is one of America's pastimes, the competitive variation of the game is also quite popular. The casual game can be played with up to four players, but competitive Scrabble is a timed one-on-one match.

Scrabble is a fascinating game from a strategy perspective for a few reasons. Firstly, players obviously want to maximize the value of their individual moves. However, they cannot squander all of their good tiles early or create openings on their board for opponents to play through valuable premium squares – as a result greedy algorithms do not work. Secondly, Scrabble is dynamically scored – players know at every moment if they

are currently winning. Finally, players have imperfect information since their opponent's rack is obscured from them and they do not know which tiles they will draw. A focal point of high level play is understanding how your opponent will *likely* act based on which tiles have already been played and which moves you will *likely* have at your disposable due to future draws. The question of how to optimally play Scrabble is one whose answer can provide insight in bringing computer algorithms closer to human behavior, using the game as a medium both can act equally on. So, that begs the question– can we make an AI that reliably beats humans at Scrabble?

1.2 Research Questions

In this paper, I will explore refining an open-source Scrabble engine called **Quackle** to increase its performance. Quackle is one of the leading AI agents for playing Scrabble based on a heuristic strategy designed by Brian Sheppard in 2002 [1]. Quackle first uses precomputed values that were tuned by game data to statically estimate the value of a word move at a given point in the game, then it simulates the future outcome of the many scenarios Scrabble's randomness can generate, and finally concludes what the best move is on average.

My research centers on two key portions of the Quackle engine. First, I explored if the rules used to govern how much to simulate potential moves can be further fine-tuned. Second, I investigated if the mathematics behind the best move determinations can be upgraded to take into account live elements of the game, like which tiles are still available in the bag for play, rather than relying on just precomputed values. In the end, I found that Quackle's rules can be changed to save time and increase performance, but that its mathematical formulas for best move calculations are fragile and require more experimentation to yield positive results.

2 Background & Related Works

While research into using computers to play Scrabble is not as popular as other board games like Chess or Go, my project is not the first to investigate it. Brian Sheppard's paper *World-Championship Caliber Scrabble (2002)* is a nucleus point in Scrabble AI research [1]. In it, he details an AI called **Maven** that can play Scrabble at a *superhuman* level, as described by Google DeepMind researcher David Silver [2]. It's effective at beating top players consistently, but still capable of defeat by skilled players [3] [4]. While Sheppard detailed the inner workings of Maven in his paper, the code was never made available to the public.

Quackle is an open source version of Maven designed by MIT Master's student Jason Katz-Brown and Scrabble Master John O'Laughlin, with some minor tweaks in how game play is done [5]. Quackle has finished ahead of Maven in tournaments and is regarded as being equal if not superior to Sheppard's original [3].

Designing a Scrabble engine is tricky. During a given turn there can be thousands of possible moves due to the size of the legal game dictionary. An element of randomness disrupts deterministic evaluation because you cannot know for certain which tiles you will draw and thus which moves will be available to you on your next turn. Quackle tackles this with a combination of robust search algorithms and time-consuming simulations.

Quackle works by first using a double-ended trie algorithm to generate all the possible moves. After Quackle acquires all the moves, it sorts the moves by their **equity**. Equity is the combination of two values: the **intrinsic** and **leave value** of the move. The intrinsic value is simply the sum of the tile **worths**, the number value assigned to a particular tile (e.g. 1 for tile A). The leave value is a custom heuristic meant to represent the value of the tiles that you chose to keep (or "leave") on your rack. So if the rack consisted of CANOEJK and CANOE was played, the leave would be JK. After studying thousands of games, certain tile combinations (**synergies**) were found to be generally more valuable than others. These generate a higher leave value, meant to capture the value of keeping strong tiles together for future use. The leave is made up of several other heuristics, like vowel-constant balancing, that create a more accurate abstraction. This calculation of equity is balancing playing a good move now with setting oneself up for later moves – moves that have high intrinsic value need to have a high leave value (the tiles you leave are worth keeping together) in order to create a high equity and be selected for play.

$$\text{equity} = \text{intrinsic value} + \text{leave}$$

This process is called **static evaluation** because the components that compose the equity calculation are fixed (or static). The tile worths, synergies and other values used in the equity heuristic are stored in precomputed tables.

The combination of perfect word knowledge and static evaluation alone is enough to

build a formidable player, but Quackle goes a step further, simulating the top set of static moves and seeing which ones generated the best future scenarios on average. Simulation consists of playing the move chosen by the static evaluator, having a computer projection of the opponent select a move in response, and then having Quackle respond with another move of its own based on the generated scenario. Since rack draw is random, Quackle repeats this process over hundreds of possible draws for each move it wants to simulate. Quackle selects the move with the highest average future win percentage, a heuristic that captures its chances of victory given the mean future **point spread** (its current score vs. its opponent's) using historical data. A move that is statically valuable will not necessarily have a high win percentage due to the moves it opens up for the opponent – simulation offsets this [4]. Quackle only simulates the top 23 static moves. This is a version of a **Truncated Monte Carlo Tree Search**, which relies on the theory that there are some paths (moves in this case) not even worth exploring due to their in-optimality – Quackle deems the top (23) static moves worthy of simulation, while the rest are discarded.

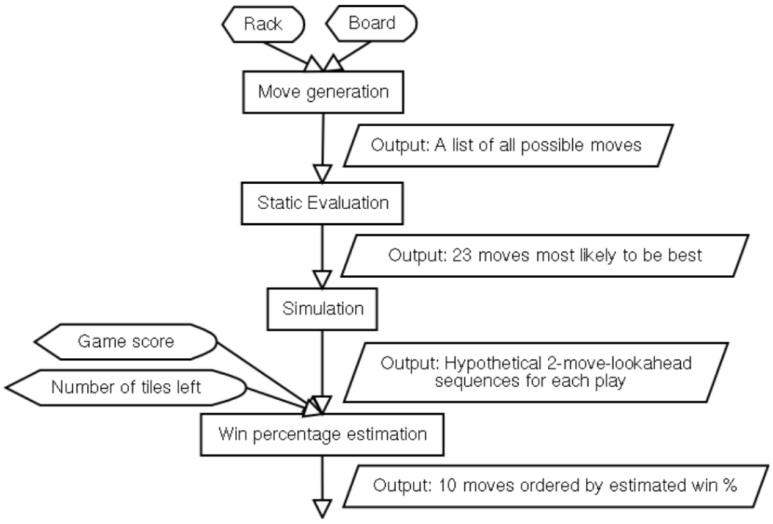


Figure 2.1: How Quackle Works [5]

Some research since then has been done into improving Quackle beyond this initial algorithm. A group out of SJSU investigated Q-Sticking, a method where you try to force the opponent to hold onto an unplayable Q or V [6]. Further work has been done investigating the possibility of leveraging **inference**, where AI players try their best to guess what an opponent has on their rack and then base their play off of these assumptions – versions of Quackle actually implement this [7]. Several collectives are also hard at work trying to apply the latest techniques in machine learning to Scrabble, designing a player that evolves as the game goes on and that dynamically learns from its losses using general models that are fine-tuned with play [8].

Despite all of this progress, Scrabble AIs are not perfect. The best humans can still beat them and the computational players find **BINGO**s (playing a word that uses all the tiles in rack, considered a highly optimal move) at a lower rate than some top professional players [3]. There is certainly room for improvement despite all the progress the field

has made.

From my research, I noticed that Quackle has no accompanying literature that is readily accessible online explaining the reasoning behind some of the values it uses for the **parameters** that controls its algorithm. Parameters create the rules of the algorithm. What amount of static moves are simulated? How far in the future should we simulate each move? These parameters control the quality of the algorithm rather than its logic, but ensuring their accuracy is necessary for confirming that each part of the algorithm is acting optimally. It is from these steps the algorithm *always* takes that I derive the name **ruled-based-heuristic**.

Furthermore, I'm skeptical of some of the rule-based heuristics being static in the first place, specifically the leave value – should it always be determined by precomputed quantities? The game itself is dynamic. Perhaps if we consider the **game state** as an input to some of Quackle's rules, it will increase the win percentage of the engine over the original version. I will explore inserting some dynamics into certain elements of the static evaluator.

I believe that refining these rule-based heuristics will lead to a stronger player and can inform the other paths of exploration other groups are pursuing.

3 Methodology

3.1 Reasoning

Here I explain the theory behind the experiments I conducted. Specifically, I justify why exploring the Monte Carlo Tree Search breadth (k) and depth (p) are important as well as why the leave value (l) could benefit from refinement.

3.1.1 k results move generation

After generating all m possible moves and sorting them by static equity, Quackle only keeps the top k results to be simulated – this dictates the breadth of the tree search. This selection of k appears to largely be arbitrary. $k = 23$ according to the the Maven paper, and Quackle shadows this, but neither Sheppard nor the Quackle authors provided solid reasoning for how this value was selected. The amount of values you choose to truncate, given by $m - k$, is incredibly important as the key assumption of Monte Carlo is that the branches you cut are unnecessary to explore. Furthermore, if I can prove empirically that the value of k can be smaller without any sacrifice to performance, that would save time on each turn that could be dedicated elsewhere. The point where a move's static equity renders it insolvent is thus critical to pinpoint, and I seek to prove it more thoroughly.

3.1.2 2-ply simulation

Once it has the k moves it wishes to simulate, Quackle does a 2-ply simulation on all k moves, which consists of the following steps:

1. Play i-th move in set of k static moves
2. Opponent player rack draws (1-ply)
3. Opponent player plays.
4. Quackle rack draws (2-ply)
5. Quackle plays
6. Increment i, repeat steps 1-6 until $i = k$

A **ply** is how many of the future rack draws Quackle chooses to do. How far in the future to simulate is critical. The Quackle authors elected to do 2-ply simulation

throughout. Theoretically, providing more plies of data to the win-percentage calculator could lead to a better player, but the trade-off is time spent conducting the plies. There is also likely a point of diminishing marginal return – the probability of the events you are examining become less and less as projections get further from reality. Again, there is no solid reasoning for the selection of a 2-ply simulation. Thus, I seek to determine the exact point where diminishing returns are met with additional plies. Determining that more plies leads to a better engine would be significant in and of itself and, similarly, determining that less plies leads to an equivalent engine would lead to turn time saved.

3.1.3 Rack Draws

When conducting simulation, Quackle conducts many rack draws. In other words, the ply simulation is done over many **iterations** and all of these feed into the move that is ultimately selected.

For each of the iterations repeat:

1. Play the i-th move in the set of k static moves.
2. Opponent player draws tiles (1-ply).
3. Opponent player plays their move.
4. Quackle draws tiles (2-ply).
5. Quackle plays its move.
6. Increment i, repeat steps 1-6 until $i = k$

Quackle assigns an amount of `secondsPerTurn` it is comfortable spending to its players and this governs how many iterations it does. If I can prove that additional time doing iterations increases Quackle's win percentage, and find time that can be saved from reducing k and the number of plies, I can assign the excess time to more iterations and unlock more performance from Quackle.

3.1.4 Dynamic Leave Value

The leave value (l) is a heuristic designed to capture the value that the tiles you choose to keep on your rack will have in the future. It balances playing valuable moves now with setting oneself up for the future. It is made up of three components and is calculated using the following formula:

$$l = \text{bt} + \text{s} + \text{vc}$$

Recall that the leave is part of the larger equation for equity:

$$\text{equity} = \text{intrinsic value} + l$$

1. **Synergy (s)**: This is a function that expresses the quality of a two-tile bi-gram based on a precomputed table. Over thousands of games, the authors of Quackle found which tiles are common in high value words and generally lead to positive outcomes together. This makes up the majority of the leave value. More positive synergies mean that the tiles go better together. Below is a synergy table that shows roughly how well the tile combinations go together. 'Y' is the best followed by '+', 'ok', '-' and 'N'. The synergies of all the bi-grams in the leave are summed together and that becomes the s that is used in the leave value.

Figure 3.1: A synergy table [9]

2. **Bad Tile Adjustment (bt)**: This is a boolean value in Quackle. It adjusts the synergy based on if a particular tile is considered undesirable. If the tile worths of the leave are below a certain threshold, the adjustment is made and the synergy is changed by the following formula:

$$s = s + 1.5 \times s - 3$$

   There is not much information on why this adjustment is made.

3. **Vowel-Consonant Balance (vc)**: Keeping a good mix of vowels and consonants on your rack is crucial since you need both types of letters to form words in the English language. This adjusts a move's value based on if the leave would saddle the player with too much of one type on their rack.

Quackle notably does not take the game state into account when making any of its decisions. This is not an oversight by the designers – Maven described weighing the board and bag as being less of priority than other elements of the game [1]. However, I believe the leave value heuristic can be improved by taking into account the current game state. My hypothesis is that if the **replacement odds** of the tiles in the leave is high, than the engine should be more comfortable playing those tiles since it is likely to redraw them anyways. We should lower the leave value, thus lowering the equity of the move, so that the tiles included in the leave are likely to be played in another move. I explored if inserting this game state adjustment to the engine led to a more capable player.

3.2 Experiment Structure

Since Quackle is open-source, I forked the repository and used it as base, editing it depending on the particular element I was testing. All tests used Tournament Word List 98, which is a standard lexicon for competitive Scrabble.

To test the effect of changes made, I had two Quackle Players play against each other over a series of 1,000 games. One player was modified with a change I was investigating and the other was the default Quackle player. I measured the win total of each of player and looked for p-value significance in either way. I also recorded the average turn lengths for each player.

Quackle is written in C++. It has a `Player` object that interacts with the Scrabble game, allowing for human and computer players.

Quackle has a `ComputerPlayer` type that inherits from the `Player` type to allow AI play. The `Resolvent` player type inherits from `ComputerPlayer` and is the AI agent.

I created a modified `Resolvent` player, titled `ModifiedResolventPlayer`, that was identical to the default `Resolvent` player except for its `m_paramters` object. To the `ComputerPlayer` type I added a field `version` to the parameters object.

This allowed for the two player objects to still share the systems necessary to execute the Quackle algorithm. Parts that were being compared had a conditional to check which player version was invoking the call and acted accordingly. This, while not being the best approach from a object-oriented perspective, balanced the fact that the Quackle codebase was not designed to have its `Player` objects use different versions of the functions that controlled the plumbing of game play; the authors anticipated all players using the same static evaluator and simulator. Creating different types of each necessary class would be time-consuming and error-prone. There were some elements that the authors made changeable between players and housed in their parameters object, which I used when applicable.

Quackle uses a `Stopwatch` class to time how many rack draws (iterations) it simulates per move. To get consistent results, I removed the use of timing and just configured each `Player` object with an absolute number of iterations. This removed a confounding factor – a player wouldn't get fewer or more iterations due to connectivity issues or otherwise.

To conduct the game series and record the result, I created a `QuackerCLI` class that runs the requisite number of games across 16 threads with flags to indicate which modifications should be made based on the version. Parallelization allowed 16 games to be run simultaneously, greatly speeding up result gathering. Experiments were conducted on the Zoo at Yale University over a VPN connection.

To see if there was a notable difference between the quality of the players, I tested the **statistical significance** of the difference between the wins of the two players. There are two quantities, n_1 and n_2, which each represent the number of wins each player has. The **null hypothesis** asserts that

$$p_1 = \frac{n_1}{n_1 + n_2}$$

$$p_2 = \frac{n_2}{n_1 + n_2}$$

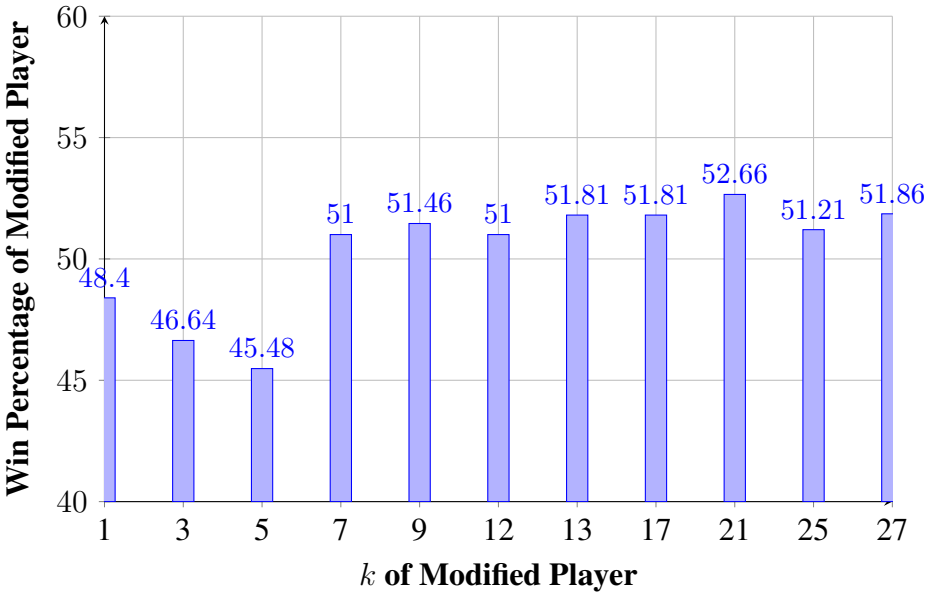
$$p_1 = p_2$$

We seek to reject the null hypothesis and test that $p_1 \neq p_2$. In other words, we seek to prove that one of p_1 or p_2 does not equal 0.5. We do this by calculating if the p-value is less than 0.05 using a binomial test. Results were gathered via CSV.

4 Results

4.1 Experiment 1 - Pinpointing k

I ran several 1,000 game series, with each series using a different k for the modified player. The unmodified player was fixed at $k = 23$, in keeping with what Maven calls for and Quackle used. This This would empirically test if changing the value of k gave an advantage to what is currently used in the Quackle engine. Note that the y-axis range is 40-60 – this is true for all barplots in this thesis.



As we see in the figure, as we increased k, the player began to see success against the $k = 23$ player. Modified players that had a much smaller value of k ($k = 1, k = 3, k = 5$) lost much more to the unmodified player. Interestingly, none of the differences reached the threshold for statistical significance. I tried a result with 2,000 game series as well, and the results were also not statistically significant and followed a similar trajectory.

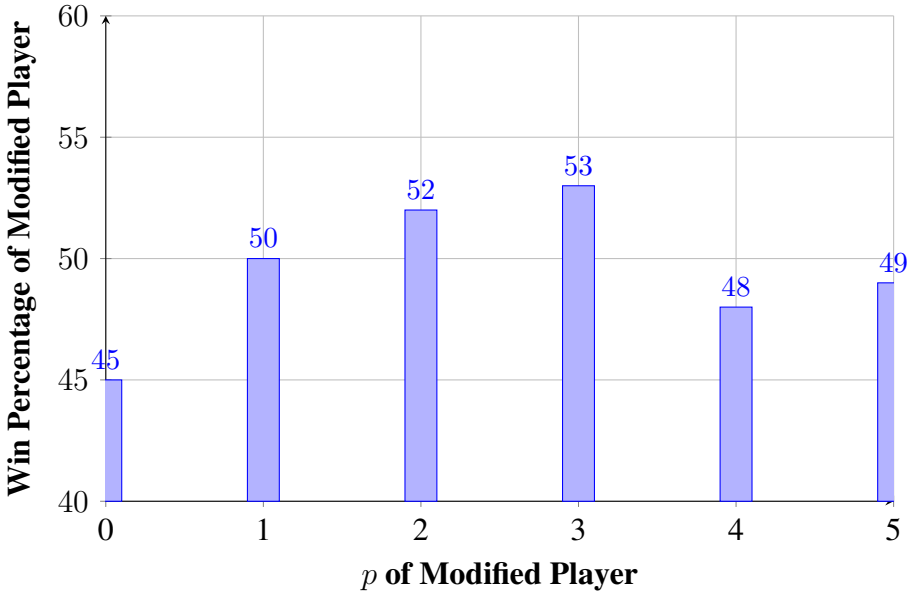
The closer k is to 0, the more static the player becomes. A static player is one that does not simulate to see the future prospects of the move. Having several moves (larger k) allows a player to choose the best move based on simulation since it has many to choose from. But a small k means the decision will largely be made based on the static equity.

It appears that static equity alone, while it leads to a worse player due to a lower winning percentage, can still hold its own against players that gain the benefit of simulation. Scrabble is an incredibly random game, which may nullify the effects of simulation. Either way, k does not need to be as high as 23 to be winningly competitive with the unmodified player. By assigning a smaller value of k, we save time during each player's

turn. The smallest value of k that won more games ($k = 7$) than the unmodified player took 44% less seconds per turn.

4.2 Experiment 2 - Pinpointing p

The next parameter I tested was the ply depth the Scrabble player used. Recall that the ply depth is how many future moves the win percentage of the current move is based on. Theoretically, deeper ply-depth should lead to a more well-informed player since it will be able to see more of the consequences of its decision. The trade off is that the point spread the calculation is based on its more improbable. In this experiment, the default, unmodified player has a fixed ply depth of $p = 2$. The modified player has its ply depth changed, with all other parameters held constant.



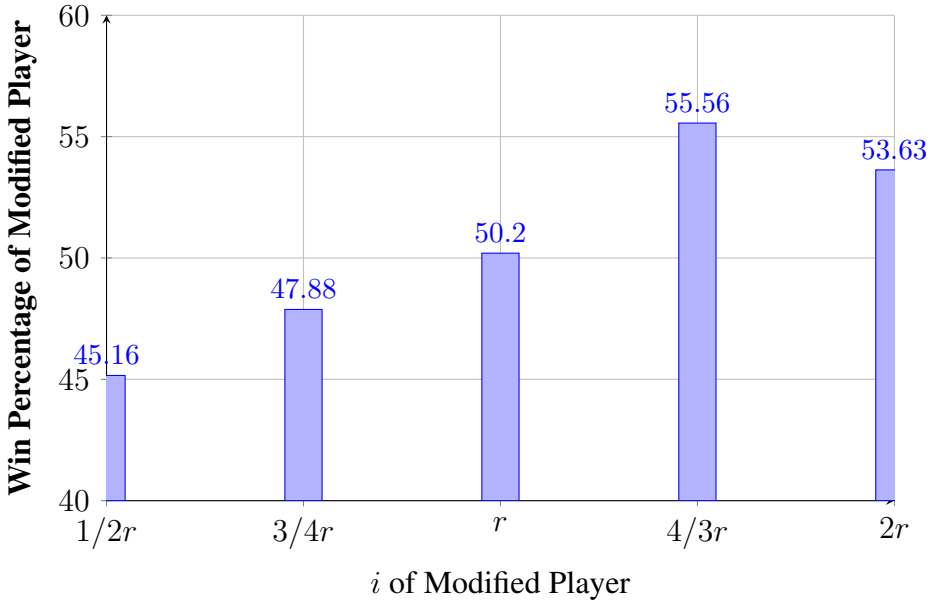
Ply depth does not have a significant influence on the capability of a player. A player with a depth of 1 is still competitive with a player that has a depth of 2. Having a depth of 5 does not make you better than a player with a depth of 2.

Interestingly, removing the notion of a ply ($p = 0$) makes the player significantly worse than a player with a depth of $p = 2$.

The presence of plies makes a difference, but the actual depth is apparently less relevant. Quackle could likely get away with using $p = 1$ rather than $p = 2$. This would save time since it would shorten the length of simulation and allow the player to run more iterations, although average time spent per turn remained fairly similar.

4.3 Experiment 3 - Rack Draws

Next, I tested how editing the number of iterations (i) a player did changed the efficacy of the player. The number of iterations the unmodified player uses is r. The modified player was fixed with multiples the unmodified player's iteration count.



Doubling the amount of iterations a player was able to do did lead to a more effective player. Decreasing the amount of iterations the modified player had access to led to a worse player. Overall, more iterations seems to allow the algorithm to converge on a more accurate "best move" since its able to take into account more potential rack draws. The trade off of these extra simulations is time – doing iterations are computationally expensive.

4.4 Experiment 4 - Designing a New Leave Algorithm

Finally, I set out to design a more effective equity calculation that factored in the current game state.

Recall that equity is calculated as the sum of the intrinsic value of the tiles played off the rack and the leave value of the tiles kept behind. Larger leave values communicate a higher equity, which makes sense since it means the move creates a better rack.

$$\text{equity} = \text{intrinsic value} + l$$

Intuitively, I felt the leave was best to target when inserting adjustments based on the game state. The leave value captures the value of keeping certain tiles together – weighting this by how probable it is to redraw those tiles may encourage the engine to play certain easily replaceable tiles sooner. It would free the engine up to play high value moves that it is previously wouldn't have due to being afraid to give up valuable, but common tiles. This could lead to a more effective equity calculation.

Let the default leave value heuristic be described by l_{default}. As described in more detail earlier, this is made up of a couple sub-heuristics that capture different elements that lead to a high leave:

1. **Synergy (s)**: Expresses the quality of that bi-gram combination of tiles, based on a pre-computed table. Over thousands of games, the authors of Quackle found which tiles are common in high value words and generally lead to positive outcomes together. This makes up the majority of the leave value.

2. **Bad Tile Adjustment (bt)**: This is a boolean value in Quackle. It adjusts the synergy based on if a particular tile is considered undesirable. There is not much information on why this adjustment is made.

3. **Vowel-Consonant Balance (vc)**: Keeping a good mix of vowels and consonants on your rack is crucial since you need both types of letters to form words in the English language. This adjusts a move's value based on if it would saddle the player with too much of one type.

As a control, I compared how a modified player with $l = 0$ (doesn't take into account the leave value) compared to l_{default}. I wished to judge how much leaves even mattered in the overall algorithm.

Statistic	**Value**
No Leaves Wins	427
Unmodified Wins	566
Number of Games	993
Δ	-139
P-value	6.88729×10^{-6}

Table 4.1: Eliminated Leaves

The result was very statistically and visually significant - eliminating the leave value worsened the player by about 139 games. Leaves are a critical part of a formidable AI player, for all of the reasons described before.

Armed with knowing the value of leaves, I designed a new heuristic to capture the game state, called the **game state adjustment**, (γ)

To calculate γ, I summed all of the weighted bigram synergies for each tile combination in the leave. Synergies were weighted by the probability of drawing each bigram combination based on what was currently left in the bag. To do this, I calculated the probability p of drawing each combination and multiplied it by the synergy $s()$ of those two tiles. The more likely they are to be drawn again (higher γ), the lower the equity goes so that the those tiles are less likely to be kept on the rack.

$$\gamma = \sum_{j_1 \in \text{Leave}} \sum_{j_2 \in \text{Leave}} p_{j_1} \cdot p_{j_2} \cdot s(j_1, j_2)$$

$$l_{\text{game state}} = s + \text{bt} + \text{vc} - \gamma$$

The goal of this, again, was the reflect the probability of drawing those tiles again in the future. γ is subtracted to the other sub-heuristics to decrease the overall equity of the move so that that the tiles that are in the actual leave are more likely to be played on the board in another move.

The results of the modified player using the new algorithm vs. the default player using the default one are as follows:

Statistic	**Value**
Game State Adjustment Wins	459
Unmodified Wins	535
Number of Games	992
Δ	-98
P-value	0.00130

Table 4.2: $l_{\text{game state}}$ vs. $l_{\text{unmodified}}$

The new leave algorithm that takes into account the game state is fundamentally worse than the default one. This result likely happened for a couple reasons. Multiplying the probability by the bigram synergy is dubious. The synergy is meant to capture the value of the tiles together, but how those values were arrived at is not specified by the authors. We only know what it is meant to capture. It is possible that the authors also took into account probabilities of future draws based on how often the tiles appeared in the bag. Secondly, working it directly into the leave is a crude approach. The leave is carefully calculated to serve a specific purpose. γ appears to be a confounding factor. Creating a third, separate heuristic that performs the adjustment without disrupting the leave might have been more effective. Overall, the results are inconclusive and more dedicated research should be done.

5 Conclusions

In this thesis, I sought to explore the accuracy of the parameters Quackle uses to govern its algorithm as well as explore if considering the game state makes a significant, positive difference in the effectiveness of a Scrabble player.

During my research, I found that Quackle parameters have some shortcomings in their tuning. I found that time could be saved, while still preserving the quality of player, by shortening ply-depth and reducing the amount of moves that are simulated. I determined that routing this saved time toward completing more iterations could lead to a more effective engine.

Furthermore, I found that attempting to make the static evaluator take the game state into account is an ineffective strategy. Static equities heuristics are very carefully tuned and trying to insert dynamic elements can skew the results.

Future work could focus on creating a more refined dynamic evaluator. Other researchers [3] and I still consider it a valuable path to explore and time constraints on this thesis limited how much I could explore it. Quackle could benefit from a thorough rewrite that makes its components more editable for future research– experiments were tough to implement and thus error prone due to the structure of the codebase. Overall, this foray into Scrabble engine research uncovered the importance of thoroughly testing MCTS assumptions and has lit a path for further research in dynamic evaluation for Scrabble.

5.1 Future Work Appendix

This project was inspired by a ”future work” section, so I seek to pay it forward. Below, I outline the aspects of my project that I believe are most interesting for further exploration in more detail:

1. Integrating all of the parameter changes I suggested into a Quackle agent and evaluating its performance would be an interesting exercise. Specifically, assigning the time saved to allow for extra iterations could provide valuable insights. Given more time, this would have been my first priority.

2. Implementing my game state adjustment (γ) as a separate heuristic is worth exploring to assess its potential impact on gameplay strategy.

3. Experimenting new manifestations of a game state adjustment besides the path I took. Other researchers and I [3] are convinced that incorporating some form of game state adjustment would be beneficial. While I attempted a version here, the approach I took is not the only one. Here are a couple of other paths for exploration:

- Recomputing the synergy table based on the current state of the board.
- Considering the probabilities of all tiles in the rack, rather than just those in the leave, when weighing the probability as part of γ.

Bibliography

[1] Brian Sheppard. "World-Championship-Caliber Scrabble". In: *Artificial Intelligence* 12.1 (Jan. 2002), pp. 241–275. DOI: 10.1016/S0004-3702(01)00166-7. URL: https://doi.org/10.1016/S0004-3702(01)00166-7.

[2] Google Deepmind. *ld28AU7DDB4*. Accessed: 2024-12-01. 2024. URL: https://www.youtube.com/watch?v=ld28AU7DDB4.

[3] César Del Solar. *Scrabble is Nowhere Close to a Solved Game*. https://medium.com/@14domino/scrabble-is-nowhere-close-to-a-solved-game-6628ec9f5ab0. Accessed: 2024-11-30. Feb. 2022.

[4] Anonymous. *Breaking the CPU: Simulation Strengths and Weaknesses*. Accessed: 2024-12-01. n.d. URL: http://www.breakingthegame.net/computers5.

[5] Jason Katz-Brown and John O'Laughlin. *How Quackle Plays Scrabble*. Accessed: 2024-11-30. 2006. URL: https://people.csail.mit.edu/jasonkb/quackle/doc/how_quackle_plays_scrabble.html#:~:text=For%20each%20future%20position%2C%20Quackle,and%20the%20number%20of%20tiles.

[6] Priyatha Joji Abraham. "A Scrabble Artificial Intelligence Game". Accessed: 2024-11-30. Master's thesis. San Jose State University, 2017. DOI: https://doi.org/10.31979/etd.uuvj-wwn9. URL: https://scholarworks.sjsu.edu/etd_projects/576.

[7] Anonymous. *Breaking the Computer: Other Features of Quackle*. Accessed: 2024-12-01. n.d. URL: http://www.breakingthegame.net/computers4.

[8] Bihan Jiang, Michael Du, and Sam Masling. *Learning a Reward Function for Scrabble Using Q-Learning*. https://web.stanford.edu/class/aa228/reports/2019/final41.pdf. Accessed: 2024-11-30. 2019.

[9] Anonymous. *Synergies*. Accessed: 2024-12-01. n.d. URL: http://www.breakingthegame.net/computers5.