# Game Playing vs. Reinforcement Learning in Othello (December 2024)

Adam Blumoff, Max Thiessen, A'Cora Hickson

***Abstract*–**This paper explores how state-of-the-art game-playing algorithms compare to reinforcement learning algorithms in the game of Othello. Our research compared Principled Variable Search (PVS) and Q-learning algorithms regarding accuracy and execution time. Our evaluations compared how these algorithms played against each other, a random opponent with a set seed (acting as a fixed opponent), and against themselves. Ultimately, the evaluations for 6000 games (1000 per match-up) suggested that PVS outperforms Q-learning significantly in accuracy against the random fixed opponent (88.9% vs. 68.1%). Also, PVS beat Q-learning most of the time with a win rate of 91.1%, doing slightly better than against the random fixed opponent. However, Q-learning took much less time to compute than PVS. Despite this, we believe that the absolute difference is insignificant (8.8 sec/game vs 0.05 sec/game). Overall, PVS is the best choice because the value of the accuracy improvements outweighs the time deficiency.

## I. INTRODUCTION

Othello or Reversi is a classic game that has been around for many years. The rules of the game are pretty simple. Othello is a two-player strategy game played on an 8x8 board where players alternately place discs, one player using black discs and the other white. A move is valid if it sandwiches one or more of the opponent's discs in a straight line (horizontally, vertically, or diagonally), flipping the sandwiched discs to the player's color. The game ends when no valid moves remain for either player, and the player with the most discs of their color on the board wins.

## II. MOTIVATION

The computational complexity of Othello has made it an appealing subject for AI research. Unlike deterministic games such as chess, Othello's dynamic gameplay involves significant uncertainty and requires adaptive decision-making. This complexity has positioned Othello as an ideal game for evaluating various AI methodologies, including search-based algorithms, heuristic evaluation functions, and reinforcement learning techniques. In addition to its utility in AI research, Othello has inspired numerous competitions and benchmarks, fostering innovation in machine learning and game-playing strategies.

Othello provides a model environment for studying sequential decision-making processes. Its deterministic nature with perfect information creates a unique context to analyze how an agent manages tactical goals in the short run and strategic objectives in the long run. The game's complexity is displayed distinctly across three primary phases: the opening game, constrained by established sequences and positional theory; the middle game, dominated by mobility considerations and positional control; and the end game, where exhaustive search becomes viable for optimal play. This phase-dependent complexity necessitates sophisticated adaptive algorithms, as demonstrated by programs like IAGO [8], an advanced Othello-playing program built around alpha-beta search algorithms combined with sophisticated evaluation functions, which implement multiple evaluation functions and search techniques tailored to each game stage [8].

## III. BACKGROUND

Although the rules and gameplay of Othello seem quite straightforward, determining the best move for each position is quite complicated. The game's computational complexity stems from several key characteristics:

Deterministic Nature: Every move follows fixed rules, with complete board visibility and predictable outcomes. This knowledge eliminates randomness but increases strategic depth. Strategic depth refers to a decision-making environment where success depends on evaluating multiple interconnected strategic factors rather than relying on probability or concealed elements [5]. With this in mind, players must consider complex factors such as mobility (potential future moves), territorial control, long-term positional stability, and anticipating and limiting the opponent's options [7].

Variable Branching Factor: The number of possible moves changes dynamically throughout the game, making applying traditional search algorithms challenging [8].

Also, 8x8 Othello is still unsolved because of the immense state space the game requires for all possible board combinations. There are $10^{58}$ possible total states where only a supercomputer (probably a quantum computer) could traverse all those spaces to make the best possible move. Exploring all these states would be possible in terms of storage but impossible in terms of traversal across all the states.

Previous research has explored various computational approaches, including traditional game-playing algorithms, Monte Carlo methods, neural network-based strategies, and reinforcement learning techniques [3]. Notably, existing top-performing engines like Edax [4] and IAGO have demonstrated sophisticated strategies, mainly using alpha-beta pruning with advanced move-ordering techniques. This research builds upon these foundations, specifically comparing Reinforcement Learning (Q-learning) with Principled Variable Search to evaluate their relative strengths in solving the 8x8 Othello challenge.

## IV. METHODOLOGY

### *A. PVS and Utility*

We chose to use PVS since the current best engine uses this algorithm. PVS uses alpha-beta pruning with iterative deepening and move ordering [8]. Previous shallower searches often determine move ordering. PVS produces more cutoffs than alpha-beta, assuming that the first explored node is the best (principle variation). Then, it tries to prove it by searching the remaining nodes with a null (scout window) faster than a regular alpha-beta window. If the proof fails, the first node isn't the principal variation, and the algorithm continues as normal alpha-beta [3]. PVS also uses iterative deepening to speed up runtimes. Instead of exploring the entire trajectory of the game tree, it only evaluates the next three moves to make an optimal decision. This selection might decrease accuracy but severely improve the game-playing process's runtime.

Another critical factor to consider is how the utility is calculated. Utility is calculated using static weights of each of the positions on the board. These weights are determined by how valuable a position is on the board. For example, the corners have the most value because they can not be flipped over and can capture a lot of discs. Although this is not the most optimal strategy in terms of accuracy since there are a lot of other heuristics to consider, due to the scope of this research, it was reasonable to use a simple heuristic and focus on algorithm implementation.

### B. Q-learning

In terms of Q-learning, the goal is to learn from playing the game over time. The state-action pairs correspond to the board state and the best possible action at that current board state. This best possible action is called the optimal policy. To avoid exploiting a policy that might not be optimal, we implemented an epsilon-greedy strategy that sometimes chooses a random move to explore more policies. We decided on an epsilon value of 0.2. Updating the q-values corresponding to the best move is calculated by taking the previous board and making a prediction about the best move based on the discounted cumulative reward of future actions. We chose a learning rate of 0.1 and a discount rate of 0.9. This reward, like PVS, is calculated using the difference between the total utility of the future and current positions. This reward is added to the error of the future predicted value of the position vs. the old and is weighted based on the learning rate of the algorithm. These state-action pairs' q values are updated every move and saved from game to game in a tabular form instead of a neural network like most state-of-the-art algorithms do today. We decided to do this because we simply did not have the time to implement this technique, but it likely improved performance overall.

### C. Algorithm Integration

When implementing these algorithms in the games, the agent makes a move using a strategy defined for each algorithm. For PVS, the strategy function simply calls the algorithm since the move is made for a given player and board. The strategy function moves for Q-learning, returns that value (since it isn't called in the function like PVS), and updates the q values. The game logic is implemented in othello.py and calls the strategy functions in the play method.

## V. EVALUATION METHODOLOGY

When evaluating the two engines, we wanted to understand how they performed in multiple situations. We would compare the two algorithms' performance against various adversarial systems. Each algorithm would play against itself (the same type of algorithm), the other algorithm, and an opponent which would make random moves. For example, when evaluating Q-learning, the three opponents it would face would be another Q-learning algorithm, the Principled Variable Search algorithm, and the random opponent. The random opponent was a benchmark because the simulation process was seeded. Therefore, the random engine would make the same move even if we ran multiple simulations. This process meant that there would be six total simulated match-ups: Q-learning vs. Q-learning, Q-learning vs. Principled Variable Search, Q-learning vs. Random, Principled Variable Search vs. Principled Variable Search, Principled Variable Search vs Q-learning, and Principled Variable Search vs Random.

For each match-up, we simulated 1000 games. For the first 500, the algorithm played as the black pieces, meaning it made the game's first move. In the following 500 games, the algorithm played as the white pieces, meaning the opposition algorithm moved first. For each game, the running win rate of the simulation and the time it took to finish the game were recorded. By the end of each simulation, these results were saved into arrays of length 1000, which were then saved into a dictionary. This data was then converted into a pandas data frame, where the data we collected for evaluation could be analyzed and compared.

There were two methods of evaluation we wanted to explore. First, we wanted to examine the win rate of each match-up. This process is a simple and effective way to judge a model's performance. In the long run, each algorithm's win rates would converge to show their true efficacy. Second, we wanted to look at the execution time of each game. Execution time was less important in evaluating an algorithm's performance, but it helped illustrate each engine's benefits (or drawbacks). We could quantify each algorithm's ability to make effective decisions under competitive conditions by examining win rates. On the other hand, execution time provided insights into the computational efficiency and scalability of the algorithms, mainly when applied to resource-constrained environments. Together, these metrics offered a balanced perspective, highlighting both the practical and theoretical trade-offs between the algorithms.

## VI. RESULTS

Below are two figures and two tables constructed with the data gathered from our 6000 simulated games. The colors associated with each match-up remain consistent across the figures and tables. In Figure 1, the running win rate for all

the match-ups fluctuates until it stabilizes after around 250 games. There is one outlier to this trend, however. In the PVS vs PVS match-up, labeled by the color brown, the agent that won every game played with the white pieces. This result means that the winner of each match always went second. In the simulation process, the first 500 games are played in black. The running win percentage stays at 0% until game 501, where it begins to rise. Table 1 shows that after 1000 games, the win rate in this match-up was precisely 50%.

Moving on to the evaluation of which engine performs best, looking at the win rate data in Figure 1, in every situation outside of the match-up mentioned earlier, Principled Variable Search outperformed the Q-learning algorithm. The best performance was in the head-to-head where the PVS engine has a Q-learning adversary. After 1000 games, the PVS engine's win rate was 91.1% against Q-learning. The best-performing Q-learning match-up was against the random opponent, obtaining a win rate of 68.1% after 1000 games. These results show the superiority of our PVS model compared to the Q-learning implementation in terms of the win rate evaluation method. The most interesting outcome was how the PVS engine performed against the random agent compared to how it performed against the Q-learning agent. It performed better versus a reinforcement learning agent than an opponent that throws out random moves. After 1000 simulations, the win rate of PVS vs. Q-learning was 91.1%, and the win rate against the random agent was 88.9% (Table 1). One possible reason for this was that the Q-learning algorithm might not have had enough data to learn from. Even though the win rates appear to converge after 1000 games in Figure 1, there appears to be a slight downward trend of the PVS model's win rate against Q-learning from game 500 onward. More time and resources allocated to the Q-learner's training might result in the match-up evening out.

**Fig. 1: Agent Win Percentages vs. # of Games Played**

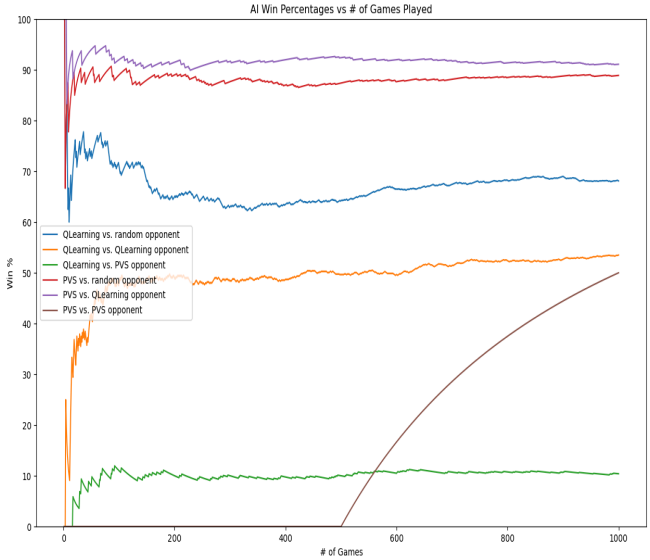


**Table 1: Agent Win Rates (%) After 1000 Games**

Agent Win Rates (%) After 1000 Games

| QLearning vs. random opponent | QLearning vs. QLearning opponent | QLearning vs. PVS opponent | PVS vs. random opponent | PVS vs. QLearning opponent | PVS vs. PVS opponent |
|---|---|---|---|---|---|
| 68.1 | 53.5 | 10.4 | 88.9 | 91.1 | 50.0 |

In terms of conclusions that can be drawn from the time data, there are no clear trends in any match-up of improvement or degradation of performance as each 1000-game simulation moves along. As shown in Figure 2, the Q-learning vs PVS, PVS vs. random, and PVS vs. Q-learning simulations varied wildly from game to game, ranging from around 0.2 seconds to nearly 8 seconds. The Q-learning vs. random and Q-learning vs. Q-learning match-ups were consistently lightning quick, while the PVS vs. PVS simulation consistently took the longest and didn't fluctuate significantly for the majority of its run time. Table 2 shows that the Q-learning agent's average time per game against the random agent was 0.028 seconds, and its average time for a game against another Q-learning agent was 0.047 seconds. The PVS engine took an average of 8.812 seconds to complete a game against another PVS agent. Overall, the Q-learning engine's execution time was much quicker in almost all situations despite having lower win rates across the board.

**Fig. 2: Agent Game Completion Time vs # of Games**

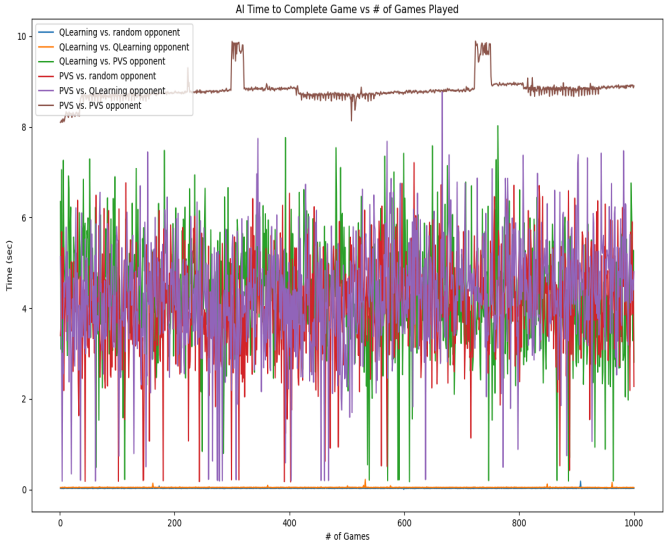


**Table 2: Agent Mean Time of Game Completion in Seconds**

Mean Time of Game Completion in Seconds

| QLearning vs. random opponent | QLearning vs. QLearning opponent | QLearning vs. PVS opponent | PVS vs. random opponent | PVS vs. QLearning opponent | PVS vs. PVS opponent |
|---|---|---|---|---|---|
| 0.028 | 0.047 | 4.275 | 4.064 | 4.267 | 8.812 |

## VII. CONCLUSION

We evaluated examples of both game-playing and reinforcement algorithms. We compared state-of-the-art algorithms like PVS and Q-learning. We tested their performance against each other, themselves, and a fixed random opponent. PVS outperformed Q-learning significantly in accuracy, but Q-learning was more time-efficient. Our analysis shows that our implementation of PVS is superior to our implementation of Q-learning because the absolute execution time improvement is not worth the decrease in accuracy.

## VIII. DISCUSSION

Although the results of our experiment appear pretty cut and dry, many decisions were made that might have affected the outcome of our PVS and Q–learning algorithms.

First, there was a decision to decrease the number of times our q values got saved and loaded. We decided to only save and load our values after every game, possibly missing some states that could have been explored. We made this choice because our simulations would crash at around 300 games. We had to make this trade-off to run a sufficient amount of games. This lack of efficiency also proves why a neural network implementation would have been a more optimal choice.

Second, we chose a static heuristic to simplify the process. Multiple heuristics could have been implemented to improve the accuracy of the algorithms, but we simply did not have the time (see Motivation and Background). Furthermore, the weights of heuristics would have to be modified during the game to ensure the highest possible performance. Also, determining the weights would have taken too long to decide what would have been optimal.

Finally, we chose not to implement epsilon decay as a part of the Q-learning algorithm. In state-of-the-art algorithms, the epsilon value decays over time as exploitation becomes more valuable. However, since it was a challenge to implement this tactic over multiple games, we decided to fix the epsilon value at its best-tested performance.

Overall, we could have made many improvements to our algorithm with more time and resources, but we were willing to make these choices to complete the research on time.

## IX. REFERENCES & REPOSITORIES


[1] M. F. Tajwar, *Othello Game AI Backend*. [Online]. Available: https://github.com/mohammadfahimtajwar/othello-game-ai-backend. [Accessed: Dec. 12, 2024].

[2] M. U. G. "How to write an Othello AI with Alpha-Beta Search," *Medium*, Jul. 17, 2021. [Online]. Available: https://medium.com/@gmu1233/how-to-write-an-othello-ai-with-alpha-beta-search-58131ffe67eb. [Accessed: Dec. 12, 2024].

[3] A. Barber, "An Analysis of Othello AI Strategies," [Online]. Available: https://barberalec.github.io/pdf/An_Analysis_of_Othello_AI_Strategies.pdf. [Accessed: Dec. 12, 2024].

[4] A. Bulmo, *Edax-Reversi*. [Online]. Available: https://github.com/abulmo/edax-reversi. [Accessed: Dec. 12, 2024].

[5] V. Muthu and M. Vaishu, "Developing Othello AI Using Minimax Search," University of Washington, [Online]. Available: https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/RUSSIA/miniproject1_vaishu_muthu/Paper/Final_Paper.pdf. [Accessed: Dec. 12, 2024].

[6] O. Zhang, *Reinforcement Learning for Othello*. [Online]. Available: https://github.com/oliverzhang42/reinforcement-learning-othello. [Accessed: Dec. 12, 2024].

[7] M. Wiering, "An Analysis of Reinforcement Learning Techniques in the Game of Othello," [Online]. Available: https://www.ai.rug.nl/~mwiering/GROUP/ARTICLES/paper-othello.pdf. [Accessed: Dec. 12, 2024].

[8] S. Russell and P. Norvig, "The Challenges of Reinforcement Learning in Game Playing," *Artificial Intelligence*, vol. 19, no. 1, pp. 123–145, 1982. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/0004370282900030. [Accessed: Dec. 12, 2024].

[9] D. Krol and J. van Brandenburg, "Developing AI Strategies for Othello," Bachelor's Thesis, University of Groningen, 2020. [Online]. Available: https://fse.studenttheses.ub.rug.nl/23036/1/AI_BA_2020_Daan_Krol_and_Jeroen_van_Brandenburg.pdf. [Accessed: Dec. 12, 2024].


**Link to Github:**
**https://github.com/adamblumoff/OthelloEngine**