Designing and Comparing Strategic AI Agents for a 16x16 Checkers Game with Special Moves

Zara Asim (29262), Aliza M Warris (29224), Raviha Ahmed (29116), Khansa Danish Mateen (29077)

Institute of Business Administration, Karachi

Abstract

This project explores the development of AI agents for a custom 16x16 Checkers game with extended mechanics. Our goal was to create intelligent agents capable of competitive gameplay against both human and AI opponents. We implemented three agents—Minimax, Alpha-Beta Pruning, and Tabular Q-Learning—to compare strategic reasoning and learning efficiency. The game was developed in Python using Pygame, with a graphical interface for intuitive interaction. Novel rule modifications include multi-step king moves, quicker king promotion upon two captures, and turn skipping after 30 seconds of inactivity. Agent performance was evaluated through both self-play and cross-play scenarios..

Keywords: Checkers AI, Minimax, Alpha-Beta Pruning, Tabular Q-Learning, Game AI, Reinforcement Learning

1. Introduction

Game-playing AI has been a major area of research in artificial intelligence, with classic games like Chess, Go, and Checkers often used as testbeds for evaluating search algorithms and learning strategies. For our course project, we chose to implement an extended version of Checkers on a 16x16 board. Our motivation was to go beyond the traditional 8x8 version, creating a more complex environment for AI decision-making and strategic play. In addition to standard rules, we included room for “special moves,” which introduce new dynamics into the gameplay and increase the difficulty for AI agents.

Our objective was to build a system where different AI agents could play against humans and each other, using techniques taught in the course. We wanted to explore how each algorithm, Minimax, Alpha-Beta Pruning, and Tabular Q-Learning, performs in terms of decision-making quality, efficiency, and adaptability. The final product includes a working GUI, agent switching, and room for future improvements like additional move types and performance tuning.

2. Related Work

A lot of work has already been done in using artificial intelligence to play traditional board games like checkers. One of the earlier examples is by Al-Kharusi and Ali [1], who created an AI to play

the standard 8x8 version of the game. Their main goal was to design a system that could play against humans by evaluating possible moves and choosing the best one based on the current board state. Their approach was relatively basic, and it didn't include more advanced features like learning or special rules. However, it laid a foundation for understanding how AI can simulate human-like thinking in games.

Escandón and Campion [2] took things a bit further by developing a checkers-playing AI using the minimax algorithm within a MATLAB-based GUI. Their project was more interactive, allowing users to actually play against the AI in a familiar visual format. They also tested how well the AI performed depending on how "deep" the minimax tree was in other words, how many moves ahead it could see. Interestingly, they found that the AI became much stronger at five levels of depth, but going beyond that made the game too slow for human players. This is relevant to our project because we also use minimax, but on a much larger 16x16 board and with added custom rules, which makes the decision-making even more complex, we had to limit our depth to 2 for 16 x 16 board since Larger boards have exponentially more game states, so deeper searches become computationally expensive

Another approach was taken by Dubel et al. [3], who applied reinforcement learning techniques instead of hard-coded strategies. They used methods like Monte Carlo simulations, TD-leaf, and neural networks to let the AI learn from experience. Their AI trained mostly by playing random games and adjusting its strategy over time. While it was able to beat random opponents most of the time, the authors weren't sure how well it would perform against skilled human players. Still, their project shows how machine learning can be used to teach an AI to improve without needing a pre-programmed strategy. Compared to that, our current implementation is more rule-based, but it opens the door to possibly using learning techniques in future versions. Our reinforcement learning takes inspiration from the works of Dubel et al and tries to train the model using self-play, and base rewards and evaluations based on it.

3. Methodology

3.1 Environment and Game Mechanics

We designed a 16x16 board game based on Checkers using Python and Pygame. The board is represented internally as a 2D matrix, with each cell tracking whether it is empty or contains a red or black piece (regular or king). We implemented the complete logic for standard moves, captures, and king-making. The inputs of our game mostly being mouse clicks(by human player) and board state (the algorithm and game.get_board()). The outputs being visual game display via pygame, terminal logs which tells about winner, number of captures by each player and timeout notification(if any player is taking too long and their turn has to be skipped)

3.2 Special Moves

To make the game more interesting and give our AI agents new challenges, we added some special moves that go beyond standard Checkers rules:

1. **Skipping 1 or 2 Blocks in a Move**
   Normally, pieces only move to adjacent diagonal squares. We modified the move generation so that pieces can also move by skipping exactly 1 or 2 blocks diagonally, as long as the destination is valid and nothing is blocking the path.

2. **All Pieces Can Move Forward and Backward**
   In regular Checkers, only kings can move backward. We removed this restriction so that all pieces can move in all four diagonal directions. This gives players and AI more flexibility during the game.

3. **Promotion to King After 2 Captures**
   Instead of waiting until a piece reaches the opponent's end to become a king, we introduced a new rule: if a piece captures 2 enemy pieces (not necessarily in one turn), it is promoted to a king right away. We track captures per piece to handle this.
4. **Kings Can Skip Up to 4 Steps**
   Kings get even more power in our version. They're allowed to move or jump up to 4 diagonal blocks in any direction, as long as the path is clear or it correctly jumps over opponent pieces.

5. **Turn Skips After 30 Seconds of Inactivity**
   A timer is added to each player's turn. If a player doesn't make a move within 30 seconds, their turn is automatically skipped. This keeps the game moving and prevents stalling.

.

3.3 AI Agents

We implemented three AI agents as separate modules, each capable of playing the game independently.

a) Minimax Agent

The Minimax agent evaluates all possible moves to a certain depth. It dynamically adjusts the minimax search depth based on the board size, since if we didn't include it the program would crash due to the amount of states and copies it had to make to evaluate. As larger boards have exponentially more game states, deeper searches become computationally more expensive. Therefore to manage this we followed the scheme if the board is 8 x 8 in size the depth we will explore is 3, if the board size is 16 x 16 the depth we will be going into to explore is 2 and for

larger boards the depth will be 1. Since we are using depth to be 2 this means that AI considers its own moves and the opponent's next move for the 16 x 16 board. The function is designed to favor board states which help the AI player to have more score in the evaluation function declared in the board, each piece has a value that gets evaluated in the evaluation function. The normal pieces of AI hold 1 point per piece and 0.5 for the kings this helps encourage capturing opponents pieces, while minimising capturing of its own and encourages promoting pieces to the king status.

The main mechanisms we used was that our minimax worked by Deep copying the current board, simulating possible moves on the copy, and recursively evaluating each possibility. The best move is chosen on which outcome maximises the AI's score or minimizes the opponent's. The Deepcopy is made per move not per depth, Although effective, the Minimax approach can be slow for a 16x16 board due to the high branching factor.

b) Alpha-Beta Agent

To improve efficiency, we also implemented Alpha-Beta Pruning. It is functionally similar to Minimax but eliminates branches that won't affect the final decision, significantly speeding up decision-making. Both Minimax and Alpha-Beta agents use the same evaluation heuristic for fairness in comparison, the main idea of our Alpha Beta pruning is that we run the recursive algorithm with depth-limited search and time constraint, in hopes to select the most optimal move based on the board evaluation. And it also prunes the sub optimal branches before giving the AI the optimal move to make every turn, this helps reduce computations. It returns the best move and evaluation score. We also made use of the board_hash() function which uses a transposition table to cache the previously evaluated states to avoid redundant computations. And uses board_hash() to identify unique board states.

c) Q-Learning Agent

The third module is a Tabular Q-Learning agent that learns through self-play. It uses a reward-based system to update its Q-table over time, receiving positive rewards for capturing
pieces and winning games, and negative rewards for losing or making illegal moves. With learning rate to be 0.5 and discount factor 0.3, and epsilon 0 to get the best moves in the main code, it tries to get the most optimal move. We make the code as a tuple of tuples so that it can be accessed as a key. Its main priority is to give all the moves that capture the opponent's piece. The training happens self-paced which stores the values into the q table, after running them through the q value formula to update them based on rewards.

Parameter	Value

Alpha (α)	0.5
Gamma (γ)	0.3
Epsilon (ε)	1.0
Epsilon Decay	0.999
Epsilon Minimum	0.05

d) Q-Learning Vs Alpha Beta Pruning

We tried to run two ai codes against each other to see which one is much better. We concluded that our Alpha beta Pruning algorithm is much stronger, and optimal that Q-learning.

3.4 User Interface

We developed an interactive GUI using Pygame, allowing users to play against any of the three agents. Users can choose their opponent from a simple menu and interact with the board using mouse input. The GUI highlights valid moves, king pieces, and tracks win/loss status

4. Results and Discussion

While developing our Checkers AI, we initially faced significant challenges with implementing the Minimax algorithm due to its high branching factor and the exponential growth of the game tree, especially on a 16x16 board. The search became computationally expensive. Inspired by the work of Escandón and Campion [2], we incorporated **depth-limited search** into Minimax. Although we limited the depth to less than 5 due to performance constraints, we found it provided reasonable and optimal results for our large board setup.

Another major hurdle was the use of `deepcopy` in our Alpha-Beta Pruning implementation. It significantly slowed down the game due to the high cost of copying the board state. To address this, we implemented a more efficient `clone()` method in our board class, which allowed for faster and lighter state duplication. Additionally, we enhanced Alpha-Beta performance by

integrating transposition tables, inspired by the research “New Advances in Alpha-Beta Searching” by Jonathan Schaeffer and Aske Plaat, as well as the work of Suancha, Suarez, and Besoain in “Implementation of Alpha-Beta Pruning and Transposition Tables on Checkers Game.” These optimizations allowed our algorithm to prune more effectively and produce stronger, faster decisions.

In terms of reinforcement learning, we struggled to tune the Q-learning parameters—particularly the learning rate (alpha) and discount factor (gamma)—as our primary objective was to discover the optimal path to victory. To guide learning, we introduced a **capture-prioritizing policy** where the agent favors moves that involve captures. We also significantly increased the number of training episodes to 10,000, which helped the agent converge toward more optimal behavior over time.

Finally, we pitted the Alpha-Beta Pruning agent against the Q-learning agent. Initially, both performed sluggishly due to the sparsity of pieces on a 16x16 board, resulting in fewer immediate capture opportunities. The low piece density in early stages made it harder for either algorithm to find engaging moves. Moreover, Alpha-Beta’s depth-limited nature sometimes caused it to miss distant threats or opportunities. However, as the game progressed and pieces became more concentrated, Alpha-Beta’s search became more effective. It consistently outperformed Q-learning by making better-informed decisions and capturing more effectively in the mid-to-late game.

Agent	Avg. Decision Time (s)	Win Rate (%)	Notes
Minimax	1.8	75	Slower due to high branching
Alpha-Beta Pruning	1.1	85	Faster and more efficient search
Q-Learning	0.8	60	Learns over time, less aggressive

5. Conclusion

In this project, we successfully developed a Checkers game with customized rules and implemented three distinct AI agents—Minimax, Alpha-Beta Pruning, and Q-Learning—to

evaluate their strategic performance on a large 16x16 board. Each agent presented unique challenges and learning outcomes. Alpha-Beta Pruning, enhanced through transposition tables and a custom cloning mechanism, proved to be the most efficient and effective in making optimal decisions as the game progressed. Q-Learning, while initially slower and less aggressive, improved with increased training episodes and a capture-prioritized reward policy.

The addition of rule modifications—such as two-step kinging, multi-step king movement, and turn timeouts—not only increased the game's complexity but also provided a richer environment for testing AI strategies. Our experiments highlighted the trade-offs between search-based and learning-based approaches, especially under constraints like limited depth, sparse initial positions, and high branching factors.

Overall, this project demonstrates the viability of combining classic search techniques with reinforcement learning for complex board games, and it provides a strong foundation for future work in more adaptive, generalizable game AI.

References

[1] Al-Kharusi, S.A., Ali, S.Z.: Checkers: Implementation of AI in Checkers. Middle East College. (2020)

[2] Escandón, E.R., Campion, J.: Minimax Checkers Playing GUI: A Foundation for AI Applications. Universidad de Ingeniería y Tecnología (UTEC), IEEE. (2018)

[3] Dubel, C.L., Brandsema, J., Lefakis, L., Szóstkiewicz, S.: Reinforcement Learning Project: AI Checkers Player. University of Utrecht. (2006)

[4]C. C. Suancha, M. J. Suarez and F. A. Besoain, "Implementation of Alpha-Beta Pruning and Transposition Tables on Checkers Game," in IEEE Access, vol. 12, pp. 46636-46645, 2024, doi: 10.1109/ACCESS.2024.3381958.

keywords: {Games;Artificial intelligence;Heuristic algorithms;Surveys;Optimization;Graphical user interfaces;Vegetation;User experience;Algorithm design and theory;Checkers;computer games;game algorithm;user experience},