Minimax and Monte Carlo Tree Search Agents for Checkers.

Author: Zachary Keyes * GitHub Repository (https://github.com/imzoc/checkers-MCTS)

Introduction.

Checkers is a two-player turn-based game with perfect information. It is a decision-making environment for artificial intelligence agents, and presents a challenge for research in the domain of game theory, decision making and artificial intelligence because of the large branching factor and strategic depth required to outmaneuver an opponent.

This paper explores the application of artificial intelligence to the game of Checkers by implementing and analyzing two types of decision-making tree-based search agents: a minimax tree search agent and a Monte Carlo tree search (MCTS) agent. Minimax tree search agents choose moves that maximize the minimum utility they expect to receive, assuming optimal play from the opponent. MCTS agents choose moves that have the highest probability of winning, based on an estimated distribution of win probabilities. Minimax tree search and MCTS will be discussed in more detail in later sections.

The primary objective of this study is to evaluate the performance and computational efficiency of these two algorithms in the context of Checkers. The comparative analysis will focus on the agents' win rates against each other to analyze performance and runtime of each agent to analyze computational efficiency. This investigation will highlight the strengths and limitations of each algorithm. It will also explore potential algorithm enhancements and hybrid strategies (i.e strategies that implement minimax tree-search and MCTS methods) in decision-making environments.

We first introduce the game of Checkers, its rules, and the starting game state. We move on to describing the Minimax agent and MCTS agent in more detail. We describe the algorithms they use, how they make decisions, and how the agents are implemented for Checkers. We then evaluate the tree-based agents' baseline performance and efficiency against a random agent (i.e. an agent that chooses randomly among its legal moves). After baseline analyses for the tree-based agents, we analyze their performance and computational efficiency against each other. Finally, we discuss how each tree-based agent can be improved using additional algorithmic techniques, as well as how minimax tree search and MCTS techniques might be combined to create an agent with even better performance and computational efficiency.

The game.

The game of checkers operates on an 8x8 board. There are two kinds of pieces: regular/"man" pieces and king pieces. Man pieces are only allowed to move on diagonals toward the opponent's side of the board. Man pieces become king pieces when they reach the last row on the opponent's side of the board. King pieces can move on diagonals in any direction. The initial state of Checkers is depicted in Figure 1.

There are two kinds of moves: single moves and jump moves. Single moves involve moving a piece one square on a diagonal to an unoccupied square. Jump moves involve moving a piece two squares on a diagonal to an unoccupied square, where the "jumped-over" square must contain an opponent piece.

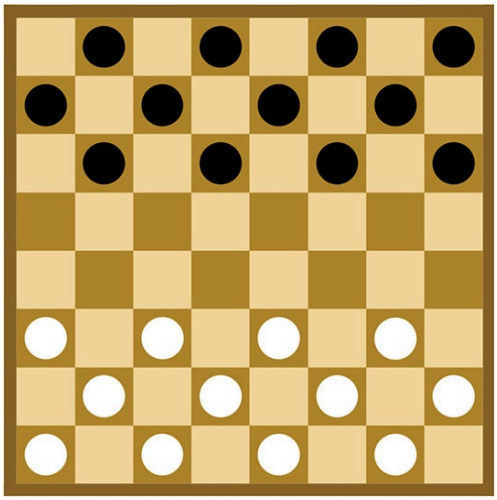

Figure 1: starting game state. Each player starts with 12 normal pieces distributed as pictured on the first three rows of their side.

Players make one move (or a sequence of jump moves) every turn. If the player is able to make a jump move, they must. After they make a jump move, they must continue to make jump moves if additional jump moves are possible. If the player cannot make a jump move from the start of their turn, they must make a single move.

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Algorithm 1: Minimax search and decision. "An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation arg $\max_{a \in S} f(a)$ computes the element a of set S that has the maximum value of f(a)." Russell & Norvig [1] (page 166).

A player wins when their opponent cannot make any moves, either because their pieces are blocked or because they have no more pieces. When a player wins, the game is over.

Minimax Tree-Search Agent.

The minimax tree-search agent uses the minimax tree search algorithm described in Russell & Norvig [1] (page 166).

In my implementation of a minimax agent for checkers, the only modification I implemented was a depth check alongside the TERMINAL-TEST base case to limit the depth of the minimax search (with a large branching factor such as that in Checkers, the number of searched states increases branch-fold with every minimax layer). The TERMINAL-TEST function is defined in the CheckersGame class by the is_game_over method. It returns whether, in the argument state, either agent has no legal moves (either because all of their pieces are blocked or because they have no pieces left). The UTILITY function is defined in the MinimaxSearchAgent by the get_player_piece_ratio method, which returns the ratio of pieces between the player the agent controls and the opponent player. This serves as a reasonable heuristic, introducing an element of very simple future reward. The ACTIONS function is defined in the CheckersGame class by the get_legal_moves method. The RESULT function is defined in the CheckersGame class by the generate_successor method.

MCTS Agent.

The MCTS agent uses a customized version of Sutton & Barto's [2] MCTS algorithm (page 185).

The Selection algorithm starts at the root of the search tree and traverses to a leaf node using a tree policy defined by the UCB1 formula (i.e. it selects the child node with the greatest UCB1 value). A leaf node in my implementation is defined as a node which does

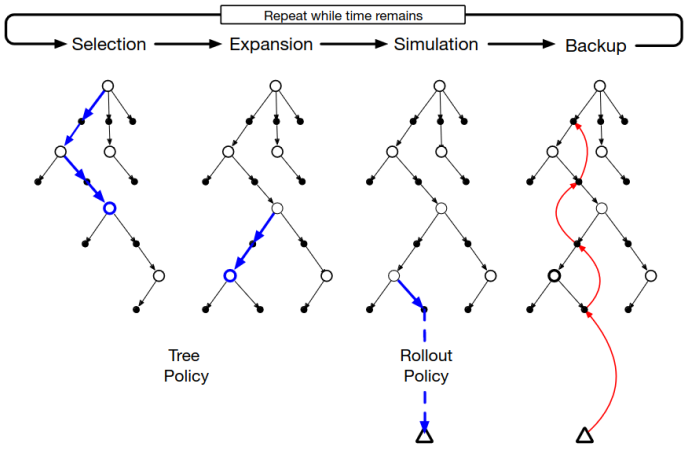

Figure 2: MCTS tree search algorithm. MCTS consists of four general algorithms: a selection algorithm traverses the tree to a leaf node; an expansion algorithm generates the leaf node's children; a simulation algorithm runs a rollout policy; and a backup algorithm backpropagates the utility from the rollout up the tree.

not have any children. It is important to note that this is in contrast to the implied definition of leaf node in Figure 2 as a node which does not have all of its children. The UCB1 formula returns an upper-confidence bound value for every state that balances each successor state's win ratio (the exploitation term) with the frequency it is visited (the exploration term).

The Expansion algorithm generates all of the state's children using the CheckersGame class's generate_successor method. The Simulation algorithm runs a random simulation (the rollout policy) from the leaf node and determines if the random simulation
resulted in a win or a loss for the agent. The Backup (or backpropagation) algorithm updates the win and visits counts for the leaf node all the way up to the root node, as well as the leaf node's successor chosen by the Simulation algorithm (if the leaf node wasn't a terminal node).

Finally, after the MCTS algorithm has been run, the agent selects an action to take from the root node. In my implementation this is the move that results in the highest win ratio, defined by get_win_probability.

Baseline analysis against a random agent.

To get baseline performance and efficiency estimates for the tree-based search agents, we will run series of Checkers games with a tree-based agent controlling one player and a random agent controlling the other player. We will discuss how agent parameters impact performance and efficiency against a random agent, on average.

For reproducibility, these simulations were performed on a Thinkpad P14s laptop with a Ryzen 5 5650U 6-core 2.3/4.2 GHz laptop CPU using Python 3.10.12. I began running the 100-game series with multiprocessing (10 out of 12 threads) about halfway through.

Baseline analysis: Minimax Agent

A Minimax agent with search depth 1 playing 100 games against a random agent results in a 92% win rate for the Minimax agent and an 8% draw rate. The average game time is 0.64 seconds with a standard deviation of 0.41 seconds. The average number of moves in the game is 71.79 with a standard deviation of 50.40. In summary, Minimax agent with a search depth of 1 has fast runtime but long games, and sometimes loses against a random agent.

A Minimax agent with search depth 2 playing 100 games against a random agent results in a 97% win rate for the Minimax agent and a 3% draw rate. However, performance is already suffering from the large branching factor: the average game runtime is 11.35 seconds with a standard deviation of 5.89 seconds. The average number of turns has decreased to 53.12 with a standard deviation of 36.67.

A Minimax agent with search depth 3 *utilizing alpha-beta pruning (discussed in a later section),*

playing 100 games against a random agent results in a 98% win rate for the Minimax agent, a 1% draw rate, and a 1% loss rate. Performance suffers from the large branching factor, but it is clear that alpha-beta pruning is helping out: the average game runtime is 19.55 seconds with a standard deviation of 10.85 seconds. The average number of turns has decreased to 51.91 with a standard deviation of 28.28.

Baseline analysis: MCTS Agent

A MCTS agent with 10 simulations playing 100 games against a random agent results in a 85% win rate, 4% loss rate, and 11% draw rate for the MCTS agent. The average runtime is 1.69 seconds, with a standard deviation of 1.02 seconds. The average number of moves in the game is 90.36 with a standard deviation of 57.11. In summary, the MCTS agent has fast runtime but has long games, and does not win about 15% of the time.

A MCTS agent with 100 simulations playing 100 games against a random agent results in a 97% win rate and 3% draw rate for the MCTS agent. The average runtime is 10.85 seconds, with a standard deviation of 5.66 seconds. The average number of moves in the game is 60.84 with a standard deviation of 40.14. In summary, the MCTS agent has moderately long runtime, decent but not perfect winrate, and still has relatively long games.

A MCTS agent with 1000 simulations playing 100 games against a random agent results in a 100% win rate for the MCTS agent. The average runtime for one game is 75.95 seconds with a standard deviation of 47.15, and each game has about 43.57 moves with a standard deviation of 11.36. Using 1000 simulations each time the agent selects a move shows a clear improvement in performance and decreases the number of game moves before the game ends, but the increase in runtime is significant.

MCTS with 10000 and more simulations should be explored, but I do not have compute for that (I estimate that simulating 100 games of a 10000-simulation MCTS agent playing against a random agent will take almost 2 hours with parallel processing).

Comparative analysis.

I now set Minimax agents against MCTS agents for 100 games. To balance performance with time constraints, I set the Minimax agent to depth 3 and made the MCTS agent run 1000 simulations.

The Minimax agent won 55% of the time. The game drawed or exceeded 200 moves (at which point I consider the game a draw) 40% of the time. The MCTS agent actually won 5% of the time, which indicates that occasionally the future predictive power of MCTS algorithm can outmaneuver the near-sighted (although heuristic assisted) perfect play of the Minimax agent.

Games took, on average, 9 minutes and 9.93 seconds, with a standard deviation of 4 minutes and 26.06 seconds. Each game had an average of 128.17 turns with a standard deviation of 63.61 turns.

While Minimax performed better, MCTS caused a draw or a loss 45% of the time, which indicates that Minimax can improve from MCTS. Hybrid strategies, where Minimax methods are implemented in MCTS, are discussed in a later section.

Enhancement techniques.

The minimax agent's efficiency in Checkers is greatly improved by alpha-beta pruning. In depth 2 search across 100 games, alpha-beta pruning improves the runtime almost 10-fold (average runtime of 1.25 seconds instead of 11.35 seconds).

The MCTS agent's performance might be improved by heuristics. However, that's beyond the scope of this project. That will be pursued in future projects, I am very sure :)

Hybrid Approaches.

Baier & Winands (2013) describe Minimax implementations in the rollout, selection, expansion, and backpropagation phases. I find Minimax implementations in the rollout and backpropagation phases to be the most interesting. I know nothing about informed rollout phases (that's also beyond the scope of this project) but I can see how an informed rollout policy could improve the MCTS-estimated distribution of win probabilities from possible actions. Minimax backpropagation is especially intriguing: if all actions taken from a game state result in a loss, it's completely unnecessary to explore any subsequent game states. The rollout policy can consider the game state a "proven loss" without playing the game out to an end state.

Conclusion.

Minimax tree search and Monte Carlo Tree Search (MCTS) are both effective algorithms against random play in Checkers. When matched up against each other, Minimax with depth 3 usually wins against MCTS with 1000 simulations, but not always.

Alpha-beta pruning greatly improves minimax tree search. Heuristic search would probably improve MCTS, but that's beyond the scope of this project. There may be potential in hybrid approaches that combine elements of MCTS and Minimax tree search.

Final Thoughts.

Who would have thought that Checkers could be such a hard game to solve? We humans play games using algorithms mostly hidden from our consciousness. Earlier today (while I took a break from this project) I was watching my favorite video game YouTuber (one of the best players in the game he plays, with over 10 years of experience and probably tens of thousands of hours) say "I feel it in my guts" to support his reasoning for making a decision. Watching someone who has practically dedicated their life to being the best player in an entire game support their decisions with "I dunno, it just feels right" is a humbling experience.

To me, this project and that moment earlier today watching my favorite video game player show the dichotomy between the human ability to perform incredibly well in complex environments and the deceptive difficulty of designing formal agents to perform well on comparatively simple problems. There is so much progress to be made.

References

[1] Russell, Stuart J., et al. *Artificial Intelligence: A Modern Approach.* 3rd ed, Prentice Hall, 2010.

[2] Sutton, Richard S., and Andrew Barto. *Reinforcement Learning: An Introduction.* Second edition, The MIT Press, 2020.

[3] H. Baier and M. H. M. Winands, "Monte-Carlo Tree Search and minimax hybrids," *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, Niagara Falls, ON, Canada, 2013, pp. 1-8, doi: 10.1109/CIG.2013.6633630.