Solving m, n, k-games: Monte Carlo Tree Search vs. Minimax

Thean Lim
Northeastern University
lim.the@northeastern.edu

Abstract

This paper explores the application of the Minimax and the Monte Carlo Tree Search (MCTS) algorithm in the context of m, n, k-games – connect k marks in an mxn board. Specifically, it is looking to uncover when MCTS should be chosen over Minimax with respect to different m, n, k, and time limits per move. A heuristic is designed and used by both agents. The Minimax agent uses Iterative-Deepening, efficient memoization, Alpha-Beta pruning, and move ordering techniques. The MCTS agent uses the UCT tree policy, max-child final move selection, and depth-two random rollout policy. The results show that the Minimax outperforms the MCTS agent slightly in some configurations. However, the relatively good performance of the Minimax agent is hypothesized to be caused by some shortcomings of the MCTS agent chosen in this paper. The biggest suspected culprit is the bad design of the heuristic function, which misled the growth of the Monte Carlo search tree. The results are inconclusive, and further experiments are needed.

Introduction

Tic-tac-toe (TTT), also known as noughts and crosses, is a two-agents adversarial game where each player takes turns to mark X or O in a three-by-three grid. The first player that connects three of his marks (X or O) vertically, horizontally, or diagonally wins. Tic-tac-toe is an example of an m, n, k-game, where the players have to connect k of their marks in an m×n grid. While a 3x3 TTT is easy for a human player to master, the game complexity increases with the board size (assuming k is well-picked, e.g., k≠1).

This paper explores and compares various algorithms' performances in the generalized m, n, k adversarial game. An agent's performance is defined as its winning rate across multiple gameplays. Furthermore, each agent will be constrained to pick a move within a time limit. Hence, its execution speed performance is implicitly baked into its winning rate - an algorithm that searches the game tree faster/deeper is more likely to win.

In this paper, a Monte Carlo Tree Search (MCTS) agent and a Minimax agent are used to playing against each other. Both algorithms are popularly used in adversarial game searching. However, running a complete Minimax algorithm on an entire game tree is time-consuming in a "real" problem because the game tree grows exponentially large. Hence, it is not uncommon to stop evaluating the game tree at a cutoff depth and estimate a nonterminal state value using an evaluation function. This causes the algorithm's robustness to depend heavily on the quality of the evaluation function. However, engineering a sound evaluation function that captures sufficient game state information is often challenging.

In addition to using an evaluation function, MCTS could solve the problem by growing the search tree asymmetrically. In essence, an MCTS agent randomly plays out the game to the end multiple times and uses the winning rates to choose the most promising game state and expand the search tree from there.

With that, this paper aims to answer the following questions:

1. In general, does MCTS outperform Minimax, or it's the other way around?
2. Does changing m, n, k, and/or time limit affect an agent's winning rate? If so, are there any noticeable patterns and possible reasons?

Background

Minimax[i, ii]

The Minimax algorithm is useful for finding an optimal move in a two-agent adversarial game. The current player is the Maximizer, and its opponent is the Minimizer; the Maximizer aims to choose a move that leads to a high utility/value game state, but the Minimizer agent does the opposite. Players take turns making a move in a two-player game, which is modeled by the alternating Maximizer and Minimizer nodes in the game tree. Each node represents a possible game state and is associated with a value/utility, assuming the opponent is playing optimally. In other words,

Code is available at: https://github.com/TheanLim/MinimaxMCTS

the Maximizer's choice of a move is conditioned on the Minimizer's and vice versa. The Minimax algorithm traverses the game tree in the depth-first search manner because the terminal states are at the bottom of the tree.

Memoization

There are multiple ways (moves taken by players) to get to the same game state (same node in the game tree) — an identical game state could appear multiple times in the search tree. Thus, a state's value calculated previously could be memoized to prevent unnecessary computation.

Alpha-Beta Pruning

Not every node in the game tree is influential in deciding which move to take. Alpha-Beta pruning aims to reduce node evaluations by cutting off useless branches. This is done by keeping track of two parameters: α and β. α represents the **largest** node/state value seen on the path to the root. The Maximizer updates α and passes it down to child nodes. Suppose at any point the Maximizer finds out the value of its child node is larger than or equal to beta, the smallest node/state value seen. It stops evaluating the remaining child nodes because its Minimizer parent has a better move (i.e., a move that leads to a lower value state). The symmetric is true for β and the Minimizer.

The efficiency of pruning is affected by nodes/move ordering. Pruning is maximized when the first evaluated child node is the "best" node/state. Note that the children's state value calculated using the Alpha-Beta algorithm might not be accurate if pruning happens.

A naive way to memoize a state's value is to use the α, β, and the state as the key. This, however, is inefficient because of the low cache hit. As mentioned by Stroetmann (2018), a better solution stores the current state value estimate and a flag stating whether the estimate is equal (no pruning), higher, or lower than the actual state value. This information is used to narrow the α and β search window if the same state is revisited.

Depth-limited

The game search tree grows exponentially large as the depth of search increases. Instead of searching all the way to terminal game states, one could stop evaluating the game tree at a cutoff depth and estimate a nonterminal state value using an evaluation function.

Iterative Deepening

It could be hard in practice to choose a cutoff depth that allows the algorithm to complete computations within the desired timeframe. One possible strategy is to run the Depth-limited Minimax Search starting with a cutoff depth of 1 and repeat the search using an incremented cutoff depth (e.g., cutoff_depth+=1) while there is time left. Since the bottom of a tree has the most significant number of nodes, the repeated computations on upper levels are not that costly.

Move Ordering

To further exploit the advantage of the Iterative Deepening technique, one could use the values calculated from the previous iteration to optimize the move ordering and reduce the game tree's effective branching factor using alpha-beta pruning.

Monte Carlo Tree Search (MCTS)[iii, iv, v]

MCTS uses random sampling of the game tree search space to grow promising branches selectively. In the context of gameplay, random sampling is also known as rollout/playout. Each iteration of the MCTS has four phases:

1. **Selection**: Traverse to a leaf node starting from the root node (the current game state) following a tree policy.
2. **Expansion**: Add a new state (child node) for each possible move taken from the current leaf node. Choose a child node following a tree policy.
3. **Simulation**: Complete a rollout from the chosen child node following a rollout policy. For example, a random rollout policy chooses/takes moves randomly forever until a terminal state has arrived. The goal is to assign a value to the selected child node.
4. **Backpropagation**: The utility gained from the rollout is used to update the nodes' information along the path from the chosen child node to the root node.

The four steps above are repeated while time (or any computation budget) is left. The MC game tree grows asymmetrically towards promising nodes, which helps reduce the branching factor.

Tree Policy - UCT

UCT is known as Upper Confidence Bound 1 (UCB1) applied to trees. It is used in the Selection phase to bias choosing promising child nodes while balancing exploration in games. One could use the following UCT expression to rank and select a child node that maximizes:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

- $w_i \,/\, n_i$ is the expected node/state value after the i −th move.
- n_i is the number of simulations completed after the i −th move.
- N_i is the parent node's number of simulations completed after the i −th move.
- c is an exploration constant, theoretically set to $\sqrt{2}$.

The expression above is evaluated to a large number when a node has a high expected value and/or is rarely visited.

Thus, it balances out the exploration and exploitation trade-off.

Final Move Selection

When the computation budget is reached, the best child is selected based on a criterion, such as:

1. Max child: the root child with the highest expected value
2. Robust child: the most visited root child
3. Max-Robust child: the root child with the highest expected value and visit counts.

Related Works

Neural Networks

The vanilla MCTS may not be efficient enough when the search space is ample (e.g., many possible moves to consider). Thus, a better policy could be introduced to guide the simulation/rollout, as well as the Selection and Expansion phases of MCTS. Ideally, the policy could encode decisions made by an expert-level player but engineering such a policy is hard. One of the solutions is to learn the expert policy using Neural Networks.

AlphaGo

AlphaGo (Silver et al. 2016) uses Supervised Learning and Reinforcement Learning to gain a tree policy, rollout policy, and a value function. The authors first used Supervised Learning to train a tree policy that predicts expert player moves well. In addition, a rollout policy was trained similarly but with the goal of faster evaluation at the expense of lower accuracy. A Reinforcement Learning network was then initialized using the gradients from the tree policy and was trained to optimize the gameplay. After that, a value function was estimated from the Reinforcement Learning network. The value function takes in a game state and returns the expected future reward.

The tree policy is used during the search to choose a child node with the highest prior probability. The child node is then evaluated using a combination of the value function and the outcome from the rollout.

This Paper

Neural Networks are not used in this paper because it is difficult to get data and train networks that output policies generalizable to a wide range of m, n, and k. Instead, a heuristic is handcrafted and used to evaluate a state.

Approach

Heuristic

A heuristic is used as an evaluation function in this paper. It is designed such that:

1. It considers all possible ways to connect k marks and assigns $1/k$ points to each mark already placed on the board.
2. It adds an additional bonus point when only one more mark is needed to win.
3. It returns a large WINNING value (i.e., 1e9) when k marks are successfully connected.

MCTS and Minimax Specifications

Please refer to the Background section for detailed explanations of the algorithms used. The same heuristic is used in both agents to ensure "fairness" when they compete.

MCTS

The UCT is used as the tree policy in the Selection phase. A uniform random rollout policy is used. Instead of playing the game until the terminal state, only two random moves (one for each player) were taken to limit randomness. The resulting state is evaluated using the heuristic. The exploration constant used is $\sqrt{2}$. In addition, the Max-child criterion is used to select the best move available to the root node.

Minimax

Alpha-Beta pruning, efficient memoization, iterative deepening, and move ordering were implemented in the Minimax agent. The heuristic is used to evaluate a state.

Parameters: m, n, k, and time limit

The parameters are m, n, k, the time limit, t (in seconds) for a move, and the chosen first player. The $m \in \{8, 9, 10, \dots, 15\}$, $n \in \{8, 9, 10, \dots, 15\}$, $k \in \{\left\lceil \frac{\min(m,n)}{2} \right\rceil, \left\lceil \frac{\min(m,n)}{2} \right\rceil + 1\}$, and $t \in \{3, 5\}$. Each agent is assigned to be the first player once for each possible configuration. One (1) point is given if the Minimax Agent wins, zero if draw, and minus one (-1) if the MCTS agent wins. The points are added up for each configuration; accumulated points reflect which agents perform better given a set of parameters m, n, k, and t. The range of accumulated points is $[-2, 2]$.

Experiments and Results

Baseline

Both MCTS and Minimax agents first competed with a Random agent, who chooses actions randomly. The Random agent always played first, and the two classic m, n, k-games (Tic-Tac-Toe and Gomoku) were selected to gather baseline performances. Both MCTS and Minimax agents won 100/100 Gomoku game simulations. As for the Tic-Tac-Toe games, both agents never lost, but MCTS won 54/100 games, and Minimax won 87/100 games.

MCTS vs. Minimax

Please refer to Figure 1. The upper and lower triangles are symmetric; hence, only the upper triangle results are plotted. There are four smaller, dotted-line squares within each larger, solid-line square. Those four smaller squares are arranged into two rows x two columns. The first row represents using $k = \lceil \min(m,n)/2 \rceil$ while the second row means using $k = \lceil min(m,n)/2 \rceil + 1$. The first column represents an experiment run using $t = 3s$ while the second column means $t = 5s$. To further clarify, the top left small square represents data/points collected from running 8,8,4-games with a time limit of 3 seconds per move. The small square to its right represents the same board but with a time limit of 5 seconds per move.

Overall, the Minimax agent performs similarly but is slightly better than the MCTS agent. The accumulated scores range from -2 to 2:

- -2 points: The Minimax agent lost both games.
- -1 point: The Minimax agent lost a game and drew with the MCTS agent in another game.
- 0 point: Minimax and the MCTS agents took turns to be the first player, and they each won a round.
- 1 point: The Minimax agent won a game and drew with the MCTS agent in another game.
- 2 points: The Minimax agent won both games.

There are no noticeable patterns on when Minimax performs better than MCTS.

Hypotheses of the Results

The relatively good performance of Minimax is likely due to some shortcomings of the MCTS agent. This can be easily seen in the baseline Tic-Tac-Toe game performances – the MCTS agent won the Random agent lesser than the Minimax agent.

The MCTS agent is hypothesized to underperform due to the following reasons:

1. Small board size and short time allowed per move. It is possible that the MCTS only gains a substantial advantage from asymmetrical tree growth with a larger board size and longer computation time.
2. Random rollout policy. Related to the first point, the MCTS agent might need a longer execution time for more simulations before the root children expected values converge. Otherwise, a more expert-like rollout policy is required. The effect of the random rollout policy is likely to be minimal in a small board size game (e.g., Tic-Tac-Toe).
3. The large WINNING value from the heuristic. Due to the random rollout policy, an opponent might not block a player from taking the final winning move. This **unrealistic** rollout causes the simulating child node to receive a large WINNING value from the heuristic evaluation function. The large value overshadows the exploration component in the UCT formula, traps the MCTS agent to keep selecting the child node, and causes the search tree to grow in an undesirable direction.

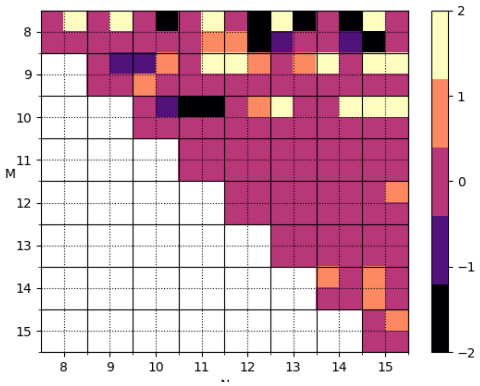

Figure 1: Heatmap showing accumulated points by the MCTS and the Minimax agents in various m, n, k, and time limit configurations. Higher points mean that the Minimax performs better.

Additional experiments should be conducted to test the hypotheses mentioned. Precisely, efforts could be placed into scaling the heuristic value within the range consistent with the reward function used in the m, n, k-game (i.e., -1 to 1).

Conclusion

The results show that the Minimax performs slightly better than the MCTS algorithm in m, n, k-games. However, the results could be misleading due to the faulty design of the heuristic evaluation function and its effect on the MCTS tree policy. Further experiments are needed to evaluate the hypotheses suggested.

References

Silver, D., Huang, A., Maddison, C. et al., 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 484–489.

Stroetmann, K., 2018. An Introduction to Artificial Intelligence. In *Playing Games*. Mannheim: Baden-Württemberg Cooperative State University.

[i] https://philippmuens.com/minimax-and-mcts.
[ii] https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/rec21.htm.
[iii] https://en.wikipedia.org/wiki/Monte_Carlo_tree_search.
[iv] https://webdocs.cs.ualberta.ca/~hayward/396/jem/mcts.html.
[v] http://www.yisongyue.com/courses/cs159/lectures/MCTS.pdf.