Connect X: Minimax Search v.s. Monte Carlo Tree Search

38879557[1]

Lancaster University Leipzig, Leipzig, Germany

Abstract. Connect X, a solved perfect-information strategy game, has established itself as a staple in game AI research. With the rise of advanced non-machine-learning search techniques like Monte Carlo Tree Search (MCTS), traditional methods such as heuristic Minimax are struggling to maintain their relevance. This study offers an empirical comparison of these approaches using both result-oriented and result-agnostic metrics to assess their performance in a Connect X setting.

Keywords: Minimax Search · Monte Carlo Tree Search · Heuristic

1 Introduction

The arrival of Monte Carlo Tree Search (MCTS) has transformed the field of game AI, notably through its integration with AlphaGo [4], offering an innovative alternative to traditional search methods for decision-making. With the aim of exploring the relative strengths and weaknesses of these approaches, this study presents an empirical comparison between MCTS and Minimax in the context of Connect X with a tailored evaluation function to facilitate it. Previous research has extensively benchmarked Minimax agents using various heuristics, demonstrating their effectiveness in games with clear rules and straightforward evaluation metrics. However, these studies often lack a direct comparison to more advanced algorithms, like MCTS, under the constraints and strategic nuances of Connect X. This study seeks to fill that gap by providing a rigorous analysis.

2 Background

2.1 Connect X

Connect X is a two-player game traditionally consisting of a board of 7 vertical slots containing 6 positions each. Both players alternate to choose a slot to drop a chip that falls to its lowest free position. The goal is to attempt to create horizontal, vertical, or diagonal chip sequences of given length X.

2.2 Minimax Search

Minimax Search, introduced by Von Neumann in 1928 [5], is considered a theoretical foundation for solving competitive zero-sum, perfect information problems [2]. The algorithm assumes two players, Max and Min, and assigns values to every terminal node in a game tree based on a heuristic or evaluation function [3]. Terminal nodes represent winning states or states at a specific depth, determined by a hyperparameter relative to the current root state. If a non-terminal node n has Max as the agent to move, the value of n will be the maximum of its successors. Conversely, if Min is the agent to move, the value of n will be the minimum. Alpha-beta pruning is often incorporated into the Minimax algorithm to evaluate the game tree at a reduced cost. It does this by "pruning" branches that are guaranteed not to affect the value of n [7].

2.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS), introduced by Kocsis et al. [8], gained prominence for automating Go [9] gameplay. As an anytime algorithm, MCTS can be stopped at any point to provide the best current estimate, often within a pre-defined budget of iterations due to practical constraints. Each iteration includes four phases, as detailed by Świechowski et al. [10]: **Selection** (explores the existing tree using a tree policy [1]), **Expansion** (adds at least one new node to the tree), **Simulation** (random gameplay to simulate outcomes), and **Backpropagation** (propagates payoffs through the tree).

3 Methods

3.1 Minimax Implementation

The Minimax algorithm was implemented recursively, with each tree node invoking the same function on its successors. Although an explicit tree data structure was constructed during the algorithm's execution, this was optional and primarily done for evaluation purposes. It is worth noting that implementation variations that do not construct this tree may yield different results for metrics like the time taken to pick a move. Additionally, alpha-beta pruning was integrated to reduce the search cost, improving the algorithm's efficiency.

3.2 Evaluation Function

The evaluation function implemented to guide the Minimax search was heavily inspired by the literature, specifically by the work of Kang et al. [6]. Although no theoretical proof will be provided to prove its status as admissible, an educated affirmation can be made through domain-specific knowledge of the game.

[1] The policy used for this study is UCB1 [1].

Table 1. Features comprising the evaluation function.

ID	**Sequence**	**Explanation**	**Utility**
1	X	Winning state.	inf
2	$X-1$	**2.1** Spaces available at both sides.	inf
		2.2 Space available at one side.	90000
		2.3 Spaces available at neither side.	0
3	$X-2$	**3.1** Spaces available at both sides.	50000
		3.2 Spaces available at one side. Depends on number of spaces available (s).	$max(0,(s-1)*10000)$
		3.3 Spaces available at neither side.	0
4	1	Depends on chip's distance to center column (d).	$max(0,200-40*d)$

The function can be broken down into 4 distinct features, as shown in Table 1. Each feature corresponds to the presence of chip sequences of length X. Moreover, each feature is comprised of sub-cases that will help to determine the value of a given game state. The most important feature is **1**, which returns a value of infinity on the winning terminal nodes, as no other state is more valuable. The function E will be calculated for the perspectives of players that maximize (max) and minimize (min), with a final value of $E(max)-E(min)$ to balance out the resulting heuristic value.

For simplicity, Figure 1 will be used to explain the subcases of feature **2** in a game with board dimensions [6, 7] and $X=4$. Assuming that red is max, $E(max)$ would yield infinity for the leftmost state, since a win is unstoppable at that point (**2.1**). The middle left state would result in 90k, as it represents an immediate advantage/threat (**2.2**). On the other hand, both the middle-right and right-most boards would yield 0, as the former cannot lead to a win, and the latter does not have any immediately available moves on either side (**2.3**). This is done to ensure that the E does not overestimate a state's utility.

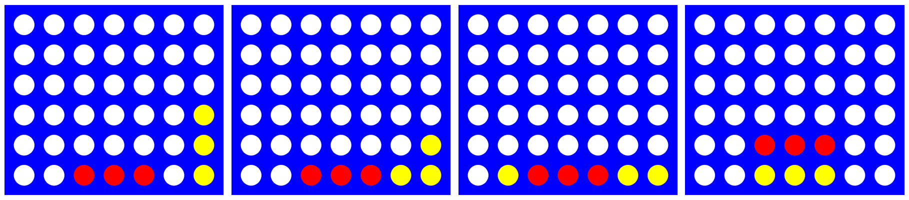

Fig. 1. Feature 2 examples.

Note that E yields a value of 0 when there is a single available space ($s=1$) on one of the sides of a $C-2$ sequence (**3.2**). The equation:

$$max(0,(s-1)*10000)$$

is proposed as it guarantees a utility of 40k for the case of 5 available spaces, the maximum of a traditional board of 7 columns, while still being lower than **3.1**. Reasonably, one could argue that this might not scale well to different board sizes, which is recognized as a weakness of feature **3.2**. In contrast, feature **4** will return a much lower value, as it is significantly less powerful than the other combinations. Chips closer to the center have a higher utility as they provide an increased number of potential plays being further from the edges of the board.

In terms of implementation, heavily optimized convolution operations were used to facilitate the detection of chip sequences of given lengths throughout the board. The convolution kernel employed consisted of all zeroes, except for the locations corresponding to the sequence being targeted, where ones were placed.

3.3 Empirical Comparison

Since Minimax search would theoretically be able to expand the entire game tree at each move to assess every possible state, four distinct “computational budgets” were set to ensure a fair comparison. The budget refers to Minimax’s game state evaluations (tree nodes) and MCTS’ game tree rollouts. Since Minimax’s budget consumption depends on its *depth* parameter, different depths are assessed. On the other hand, since the number of MCTS’ rollouts is budget-agnostic, three different strategies were proposed to control its consumption behavior:

- **Thrifty:** Allocates the same budget to every move, assuming a game with $rows \times cols$ chips dropped.
- **Optimistic:** Allocates a budget of 1 to the last move, with uniformly increasing budgets for earlier moves. This agent is *optimistic* about the possibility of winning earlier with a larger budget.
- **Greedy:** Allocates the entire budget equally among the first half of potential moves, leaving no budget for the second half.

When an agent runs out of computational budget in the middle of a game, it forfeits, and its opponent automatically wins. Note that Minimax will have *depth* number of additional moves after its budget is consumed, since it can reuse the tree computed in previous moves without further state evaluations.

Two main simulation sets were run to empirically compare both algorithms. Sets consist of 12k and 19k total simulations, respectively. Both involved running a number of repeats per unique combination of hyper-parameters to ensure fairness in the assessment of both agents. The hyper-parameters in question were, **first set**: *depth*, starting agent (to get rid of a possible source of bias), budget, and MCTS strategy; **second set**: board dimensions (and X values)[2], budget, MCTS strategy, and starting agent. Additionally, depths were not included in the second set as only the best-performing ones were selected to compete (results from the first).

[2] For every board with column number c, the rows were $c - 1$ and X was $c - 3$, to maintain the original Connect 4 proportions.

For every move, relevant metrics and game attributes were recorded in a parquet file that was later used for the analysis. Some of these metrics included: time (ms), number of tree nodes, current agent, etc.

4 Results

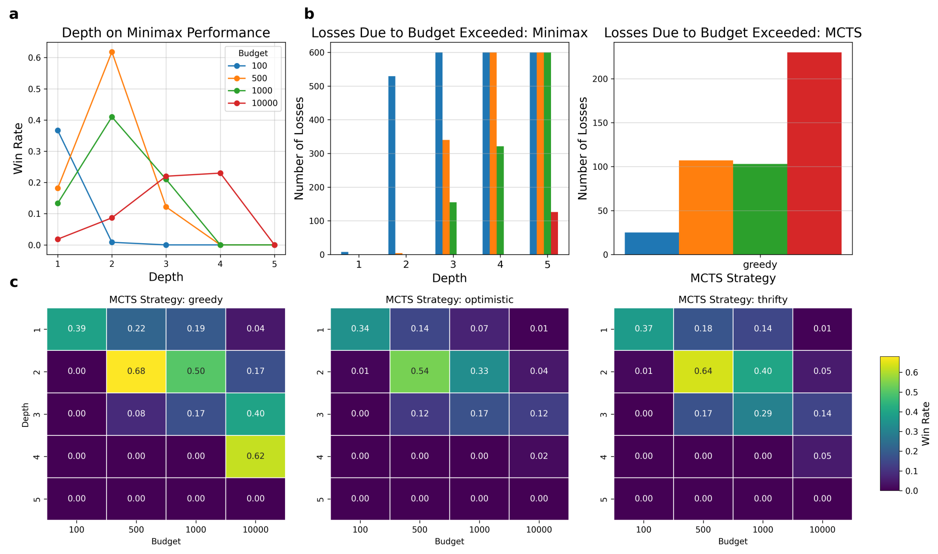

Fig. 2. **a**,**b**,**c**, Analysis of game results. **a**, Line plot of the effect of Minimax's depth on its (adjusted) win rate across multiple budgets. **b**, Bar plots of number of losses due to forfeit/budget depletion across multiple budgets. Color scheme follows that of **a**. Left: Comparison of Minimax depths. Right: Greedy MCTS. **c**, Heatmaps of Minimax's win rate across multiple depths, budgets and MCTS strategies. Each map represents a strategy. Color scheme maps win rate.

The first set of simulations determined the best-performing Minimax depth at each budget: 100:1, 500:2, 1k:2, and 10k:4. This was estimated by assessing the win rate and the number of losses due to budget depletion, as shown in Figure 2. In the case of MCTS, the only strategy that could suffer from this issue was the greedy approach, which, interestingly, performs "worse" at higher budgets. Minimax underperformed MCTS at all budgets except for 500, with a win rate exceeding 50%.

In terms of result-agnostic metrics, the memory consumed and time taken per move were recorded. Figure 3a illustrates that Minimax agents with the best-performing depths took significantly longer to make a move at lower budgets, with the time difference diminishing for budgets of 1k and 10k. However, MCTS exhibited greater variance for all budgets except 10k. Additionally, a clear pattern emerged regarding the relationship between Minimax's depth and time. This is evidenced by the difference in time between budgets 500 and 1k ($depth = 2$) being almost negligible. Moreover, a noticeable jump in time occurred for the 10k budget ($depth = 4$) for both agents, with mean times being approximately 10 times higher. Figure 3b shows that the mean memory consumption at budget 1k was significantly higher for MCTS. As was the case with time, the variance was also higher for all budgets except 10k.

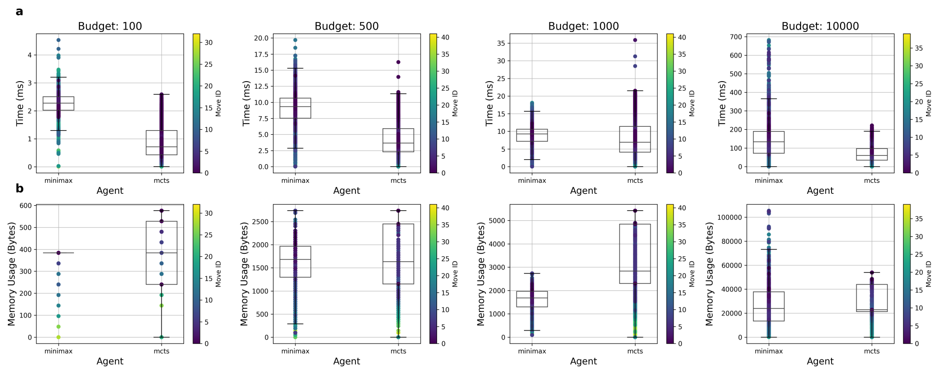

Fig. 3. a,**b**, Visualization of result-agnostic performance metrics. Only Minimax's best depth at each budget. Color scheme represents the timing of each move, indicating how early or late in the game it was performed. **a**, Box plot visualization of time (ms) taken to pick a move across all budgets. **b**, Box plot visualization of approximation of memory (bytes) consumed at each move. Calculated as $mean(MinimaxNode(bytes) \div MCTSNode(bytes))$.

Figure 4a shows that Minimax outperformed MCTS (win rate over 50%) at three new branching factors: one at the 100 budget and two at the 1k budget. Surprisingly, for each of these combinations, Minimax had a win rate of over 60%, with the two corresponding to the 1k budget approaching 70%. For the 10k budget, however, the win rate remained close to 0% for every branching factor except for 7.

For clarity, the combinations of budget and X values will be expressed as (B, X). From Figure 4b, it is evident that there were few Minimax wins at the 10k budget, with no wins at all for (10k, 5) and (10k, 7). Whereas the highest number of wins occurred at (100, 3), (100, 5), (100, 7), (500, 3), and (1k, 5-6). Moreover, the only cases with a significant difference in the average number of moves to reach a win (over 2 moves) favored Minimax, specifically at (100, 5),(100, 7), (500, 56), (1k, 5-7). The opposite occurred at (10k, 3) and (10k, 6-7); however, there were not enough Minimax wins at those combinations to properly assess this.

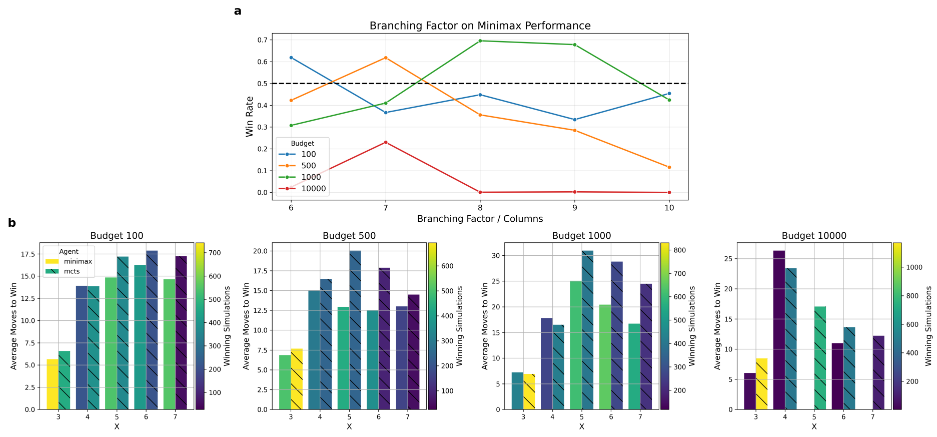

Fig. 4. a,**b**, Simulations extended to variable board/rule configurations. Again, Minimax's depths at each budget were chosen according to the previous analysis. **a**, Line plot visualization of effect of game's branching factor (board's number of columns) over Minimax's win rate across multiple budgets. The dashed line marks the 50% win rate, with every point above it indicating performance superior to that of MCTS. **b**, Bar plots showcasing the average number of moves played to reach a win across X values. Every plot represents a budget, the color scale indicates the number of simulations taken into account to compute the average and the hatch indicates the agent.

5 Discussion

As evidenced from the results, the number of losses due to budget depletion is inversely proportional to the win rate for any given budget, which aligns with theoretical expectations. Figure 2c reveals that the optimistic MCTS performed the most consistently across multiple budgets and depths, followed by the thrifty and greedy. The heatmaps reveal that the choice of MCTS strategy is crucial. Although the idea of a greedy agent was appealing, particularly for larger budgets, games tend to take longer when agents can afford to utilize more computation, which resulted in a significant number of greedy losses due to budget depletion. On the other hand, it is pleasing to see that the optimistic agent performed relatively close to expectations. Since the game tree is significantly bigger at early stages–as will be explained later–the higher initial budget allows for MCTS to build a better foundation of chips.

Figure 3b shows that picking moves at later stages of the game requires less memory for both algorithms, which aligns with a decreasing branching factor for the states in question. However, no clear patterns emerged concerning the time taken to pick a move and its timing–how late or early in the game the move was played. Moreover, MCTS built overall larger game trees in most budgets, which could explain its higher win rates. In fact, the only case where Minimax's mean memory consumed was significantly higher (budget 500) is also the case where Minimax could win more games than its Monte Carlo counterpart. One could think of this budget and *depth* combination as a "sweet spot" in Minimax's favor, where the richest trees were able to be generated without risking computation depletion. Moreover, the growth in time and memory is exponential with respect to *depth* and budget for both agents.

For the second set of simulations, one could state that MCTS generalized better across different board and rule configurations. However, it was surprising to see Minimax achieve really high win rates for some of these new combinations. Focusing on the three cases mentioned before, one could transfer the information gathered from the result-agnostic box plots to affirm that increasing the branching factor allowed for a Minimax algorithm that fully utilized its computational budget (since a higher branching factor would result in bigger game trees), specializing in outclassing MCTS for said combinations (around 70% win rate). Moreover, one could state that Minimax performs better at really low budgets and small boards, since terminal states are reached faster. On the contrary, MCTS shines at higher computational budgets, particularly when paired with larger board sizes. Interestingly, the weakness identified on feature **3.2** did not really hold in practice, as budgets 100 and 1k didn't appear to perform significantly worse for bigger board sizes.

It is safe to say that Minimax is more efficient with its wins, as it is shown that fewer numbers of moves are needed to end a game. This could be interpreted as E being either really accurate of a game state's utility or, alternatively, relatively poor.

6 Conclusions

MCTS proved to be generally more effective in terms of time and memory consumption. Note that, in this case, higher memory usage is desirable as it indicates that the computational budget is being utilized to its full potential. As demonstrated by the data collected, MCTS was also the superior search algorithm in terms of game results, with some interesting exceptions.

A limitation of this study lies in the decision to include all MCTS strategies in the analysis. It is reasonable to think that this approach may not have been entirely fair to MCTS, as the optimal Minimax depths were tailored for each budget. This choice was motivated by a conflict between the optimal Minimax depth and the best-performing MCTS strategy, as these configurations rarely align. Moreover, while Minimax depths were strategically selected to minimize defeats caused by budget depletion, this risk remained present, particularly with bigger board sizes. On the other hand, selecting a specific MCTS strategy that avoids such losses (e.g., thrifty or optimistic strategies) could be perceived as unfair to Minimax.

Future extensions of this study could involve exploring alternative MCTS strategies or refining the evaluation function. Otherwise, another direction might be redefining the budget's nature to reflect more realistic measures, such as time allowed to pick a move or CPU utilization.

References

1. Auer, P.: Finite-time analysis of the multiarmed bandit problem (2002)
2. Beal, D.F.: The nature of minimax search. Institute of Knowledge and Agent Technology (1999)
3. Campbell, M.S., Marsland, T.A.: A comparison of minimax tree search algorithms. Artificial Intelligence **20**(4), 347–367 (1983)
4. Gibbs, S.: Google's alphago seals 4-1 victory over grandmaster lee sedol. The Guardian (2016),
5. Holler, M.J.: John von neumann (1903–1957). In: Handbook on the History of Economic Analysis Volume I. Edward Elgar Publishing (2016)
6. Kang, X., Wang, Y., Hu, Y., et al.: Research on different heuristics for minimax algorithm insight from connect-4 game. Journal of Intelligent Learning Systems and Applications **11**(02), 15 (2019)
7. Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning. Artificial intelligence **6**(4), 293–326 (1975)
8. Kocsis, L., Szepesv'ari, C.: Bandit based monte-carlo planning. In: European conference on machine learning. pp. 282–293. Springer (2006)
9. Shotwell, P.: Go! More than a game. Tuttle Publishing (2011)
10. 'Swiechowski, M., Godlewski, K., Sawicki, B., Ma'ndziuk, J.: Monte carlo tree search: A review of recent modifications and applications. Artificial Intelligence Review **56**(3), 2497–2562 (2023)