MNK Game

Wadood Alam*, Matthew Pacey*, Joe Nguyen* (Equal contributions)

June 4, 2022

1 Introduction

In this paper we study the m, n, k game, in which two players take turns to place their symbol (either a circle or a cross) on an m x n grid. The winner is the player who first gets k consecutive symbols either vertically, horizontally, or diagonally. Tic-tac-toe is an example of a 3, 3, 3 game. We evaluate two search algorithms playing against each other: Minimax with Alpha-Beta (AB) pruning and Monte Carlo Tree Search (MCTS). Results will be analyzed in section 4 and some insights will be discussed in section 5. Our experimentation and data indicate that given the heuristic ordering of alpha-beta and the nature of MCTS, dependency on iterations, Alpha beta proved to be dominant force but this is likely due to the limitation placed on MCTS(discussed later in the paper). The source code is published at this Github link.

2 Search Algorithms

2.1 Monte Carlo Tree Search

The Monte Carlo Tree Search operates using a number of simulations of various move selections to identify moves that will likely lead to wins. The algorithm operates on a tree where the root node represents the board configuration (occupied and empty cells) for the current player. Each expanded node represents an empty square that the player can take. Each iteration of the algorithm does the following operations:

2.1.1 Selection

Select the node with the highest value, calculated using the Upper Confidence Bound applied to Trees (UCT) algorithm. This algorithm balances exploitation (nodes with higher win values) and exploration (visiting new/infrequently visited nodes). The motivation is that a node with a high win percentage now may not lead to a winning configuration later, hence why exploration is key to UCT. In Figure 1, the leftmost node off the root has the highest UCT value (0.98), so it will be selected.

2.1.2 Expansion

The selected node is expanded by picking a child node to expand using a policy/heuristic.

Heuristic expansion The heuristic we used was to pick the empty cell with the most occupied neighbors with the idea being that a cell with occupied neighbors would be valuable to either player (either this continues a sequence for the current player or stops a sequence for the opponent). If multiple empty cells have the same occupied neighbor count, the policy selects one of them randomly. The node is then expanded by placing the players mark (circle or cross) in the corresponding cell on the board. In Figure 2, there are two occupied cells (0 and 1) with seven empty cells. The heuristic would identify cells 3 and 4 as ideal candidates because they each have two occupied neighbors (0 and 1).

2.1.3 Rollout

Once the current player has selected a cell (in expansion), the remainder of the game is played out by selecting random empty cells for both players until a terminal state of the game is reached (win, loss, or tie). A value is assigned to each result; a win for the current player is worth one point, a tie is worth half a point, and a loss is worth zero points.

2.1.4 Backpropagation

The result of the rollout is fed to the parent of each node in the chain all the way back to the root. Each node keeps track of score (half a point for a tie, one point for a win) and number of games played. These two values are used in the UCT algorithm to calculate each node's value. In Figure 1 the result of the rollout was a loss, so the backpropagation reduced the UCT value of the initially selected node from 0.98 to 0.95.

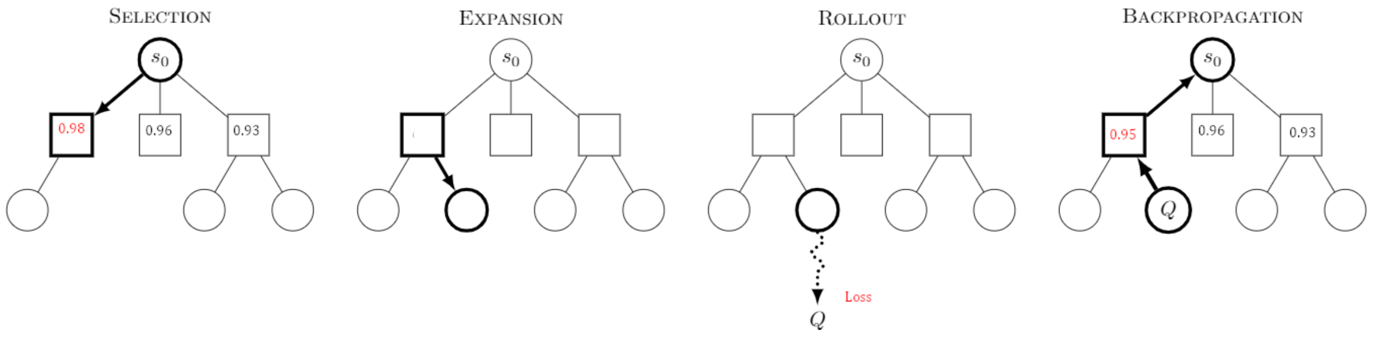

Figure 1: Example MCTS Tree

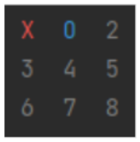

Figure 2: Example Board (3x3 with two moves played)

This cycle repeats for as long as the MCTS algorithm is allowed to run, refining the UCT values of each node with each cycle. For our setup, we determined that 500 iterations of MCTS per each move calculation was a good value of win utility versus compute performance.

2.2 Minimax with Alpha-Beta Pruning

The Minimax Algorithm operates by attempting to maximize the utility (win percentage) for the given player while minimizing the utility of its opponent.

For the MNK Game, this is accomplished by creating a tree representation of the valid moves for a given board (empty squares that the player may take). The entire search tree for boards larger than 3x3 will get exponentially large, so complete evaluation is time prohibitive. The Alpha-Beta algorithm assists Minimax by pruning (not evaluating) branches that do not increase the utility for the current player. The 'max' in the name refers to the current player, it wants to maximize its chances of winning; the 'min' refers to the player's opponent, it wants to minimize the player's chance of winning (thereby maximizing its own chance). The initial value for alpha and beta are initialized to worst case values for each player (alpha is negative infinity, beta is positive infinity). For max nodes (current player), this means pruning branches where the node value is greater than or equal to beta; for min nodes (max's opponent) this means pruning branches where the node value is less than or equal to alpha. The pruned branches in either case represent paths that do not improve play for either opponent, hence they are discarded. An example of Minimax tree search is illustrated in Figure 3.

Heuristic ordering Similar to section 2.1.2, we sort the order of an empty cell move in descending order based on the number of occupied adjacent cells to an empty cell. Our intuition is that the better move should come first so the algorithm can refine alpha and beta, and thus worse moves can be pruned out given the refined knowledge so far about the best range of backed-up value.

3 Experimental Setup

3.1 Board Configurations

We tested the algorithms with following board configurations (mxn):

- Square - 3x3, 4x4, 5x5, 6x6, 7x7
- Rectangular - 3x4, 4x5

3.2 K Values

We tested with k ranging from 3 to the size of the smallest board dimension. For a 4x5 board, this means we tested k at 3 and 4. For larger boards, we stopped k at 5 because those boards primarily led to ties.

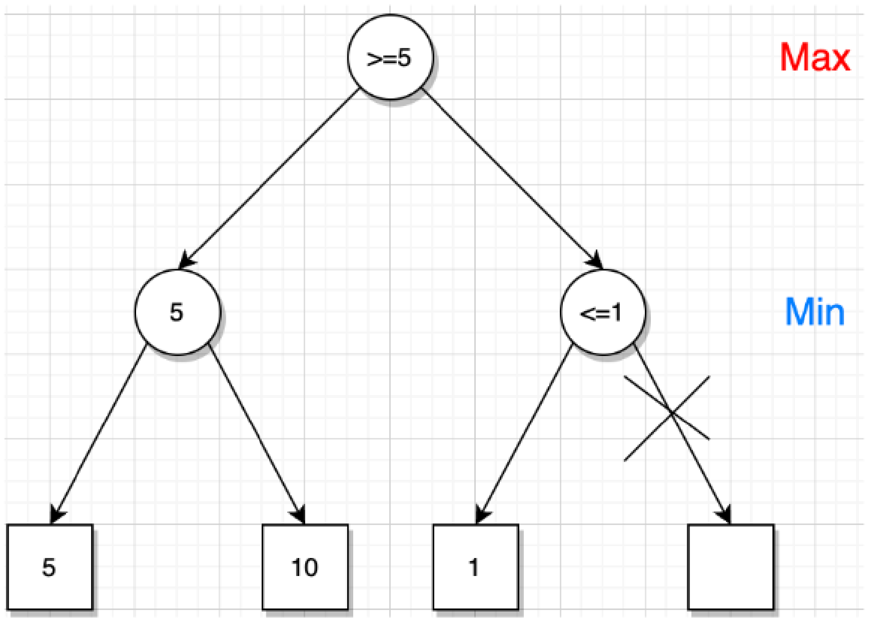

Figure 3: Example of Alpha-beta pruning in Minimax

3.3 Match-ups

We tested the following match-ups ($Player1$ vs. $Player2$):

- Alpha-Beta vs. MCTS
- MCTS vs. Alpha-Beta
- MCTS vs. MCTS (with varying MCTS iteration levels per player)

The motivation behind the MCTS vs. MCTS matchup was to demonstrate how much of an impact the MCTS iteration limit can have. Each player was allowed either 5 or 500 MCTS iterations per move selection.

4 Results

We first consider the results of each search algorithm in section 4.1, then we examine the results for games where two search algorithms compete with each other in section 4.2.

4.1 Analysis of each search algorithm

4.1.1 Minimax with Alpha-Beta

We run minimax with alpha-beta prunning and heursitic ordering with no depth limit. Numbers of nodes expanded with and without heuristic ordering are presented in Table 1.

Size of the game	Number of nodes expanded		Percentages compared to without
	Without heuristic ordering	With heuristic ordering	
3 x 3 k = 2	165	26	15.75%
3 x 3, k = 3	18297	536	2.93%
4 x 4, k = 2	813	47	5.78%
4 x 4, k = 3	1024394	4242	0.41%!!
4 x 4, k = 4	Too long to run	Too long to run	Too long to run

Table 1: Minmax with Alpha-Beta and heuristic ordering: we generate all the tree (without any depth limit)

4.1.2 MCTS

Player 1 Wins	Player 2 Wins	Player 1 MCTS Iterations	Player 2 MCTS Iterations
60	40	500	500
45	55	5	500
85	15	500	5

Table 2: MCTS vs. MCTS with different iteration counts

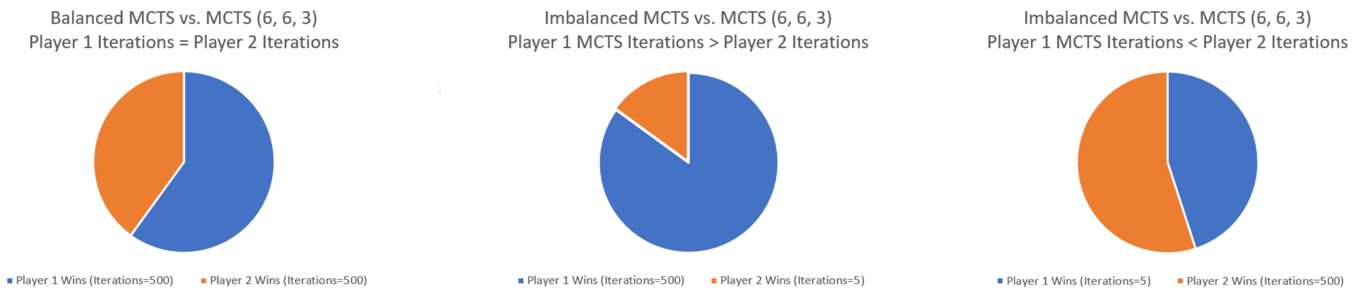

Figure 4: MCTS vs. MCTS with varying iteration count

We want to examine how the number of iterations in MCTS affects the final performance. Therefore, we conduct a game between two MCTS players and assign different levels of iterations to each player disproportionately. The details can be found in Table 2, Figure 4 illustrates numbers from that table in pie chart for better visualization.

4.2 Games between two search algorithms

We run two match-ups: 1) MCTS vs AB and 2) AB vs MCTS according to set up explained in section 3. Details can be found in Table 3.

Grouping K For better visualization in the trend when we increase the complexity of the problem, we fix the value of K: $K = 3$ and $K = 4$ in two match-ups. Details of match-ups between AB vs MCTS and MCTS vs AB can be found in Figure 6 and Figure 5, respectively.

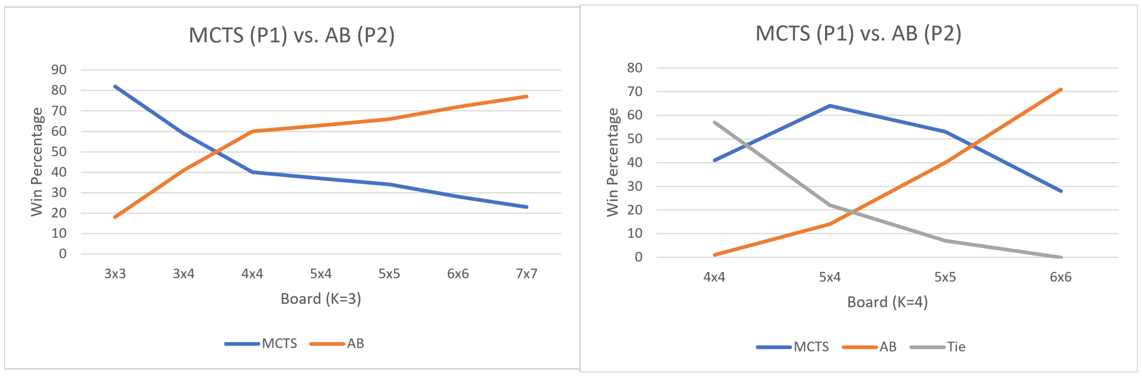

Figure 5: MCTS (Player 1) vs. AB (Player 2)

M	N	K	Player 1	Player 2	P1 Win %	P2 Win %	Tie %	P1 Avg. Time/Move	P2 Avg. Time/Move
3	3	3	MCTS	AB	82	18	0	0.1628	0.0040
3	4	3	MCTS	AB	59	41	0	0.2322	0.0070
4	4	3	MCTS	AB	40	60	0	0.3505	0.0170
4	4	4	MCTS	AB	41	1	57	0.3259	0.0937
5	4	3	MCTS	AB	37	63	0	0.4638	0.0416
5	4	4	MCTS	AB	64	14	22	0.4141	0.2056
5	5	3	MCTS	AB	34	66	0	0.6599	0.1032
5	5	4	MCTS	AB	53	40	7	0.5668	0.5050
5	5	5	MCTS	AB	13	0	87	0.5383	0.4691
6	6	3	MCTS	AB	28	72	0	2.4313	1.4468
6	6	4	MCTS	AB	28	71	0	2.2983	1.5313
6	6	5	MCTS	AB*	30	28	42	1.9314	1.1721
7	7	3	MCTS	AB	23	77	0	1.9761	4.1244
3	3	3	AB	MCTS	30	70	0	0.0079	0.1490
3	4	3	AB	MCTS	76	24	0	0.0219	0.2156
4	4	3	AB	MCTS	79	21	0	0.0577	0.3195
4	4	4	AB	MCTS	4	63	33	0.1116	0.2880
5	4	3	AB	MCTS	88	12	0	0.1386	0.4506
5	4	4	AB	MCTS	10	80	10	0.2309	0.3748
5	5	3	AB	MCTS	82	18	0	0.2928	0.6045
5	5	4	AB	MCTS	23	74	3	0.5017	0.5219
5	5	5	AB	MCTS	0	23	77	0.5125	0.5356
6	6	3	AB	MCTS	90	10	0	7.0012	2.2287
6	6	4	AB	MCTS	47	52	1	1.4993	1.9524
6	6	5	AB*	MCTS	9	55	36	1.2548	1.7440
7	7	3	AB	MCTS	94	6	0	38.4386	1.5269

Table 3: AB vs. MCTS Matchup Statistics; AB* denotes runs where max depth was set to 4 (6 for all other runs)

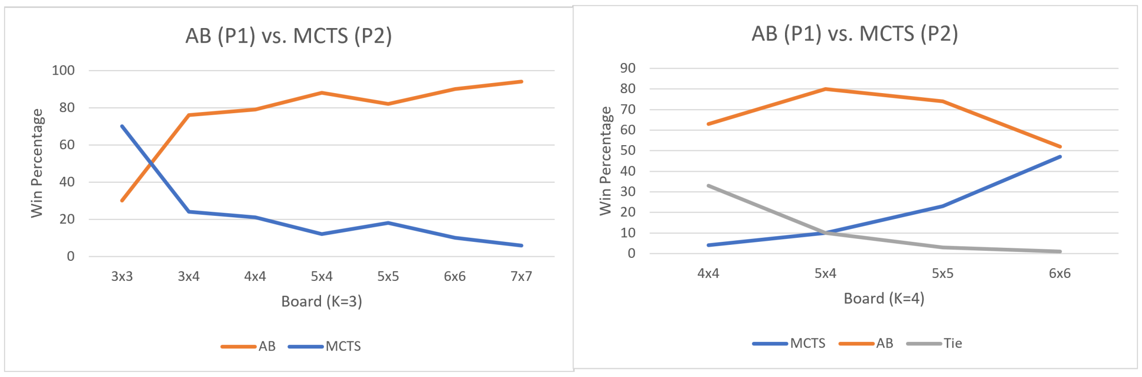

Figure 6: AB (Player 1) vs. MCTS (Player 2)

5 Discussion

We reached the following conclusions:

5.1 Overall insights about the game

Nature of the game Player 1 (first player to select a square) has an advantage. This is intuitive but was demonstrated in Figure 4) where the first player to move had a win advantage unless the player was severely handicapped by limiting the MCTS iterations.

Values of K Given Table 3, Figure 6, Figure 5, Higher k values (5 or more) led to mostly tie games. In small sample runs, when k was more than 5 the majority of the matchups were ties, so that data is not included.

5.2 Match-up between two search algorithms

MCTS vs AB Given the Table 3, for a smaller boards (3x3), MCTS was able to outperform AB. However for boards larger than 5x5, AB was able to outperform MCTS even when MCTS had the advantage of playing first. This can likely be attributed to the relatively lower MCTS iteration limit to keep its move selection fast. If the iteration limit was increased, MCTS may have done better or at least forced more ties. Given the nature of our parameter restrictions, Heuristic ordering, alpha beta turned out to be a dominant force.

5.3 Each search algorithm

5.3.1 MCTS

Iteration in MCTS Given Table 2, increasing the number of MCTS Iterations increases win percentage. The optimal value for smaller boards seemed to be 500 iterations (number of playouts to simulate for each move decision), past that not many wins were gained at the expense of longer compute time. The win percentage suffered at more complex boards, further improvements could be made by increasing the iterations for bigger boards.

5.3.2 Minimax + Alpha-Beta

Depth limit in Minimax + Alpha Beta Keeping a consistent Alpha-beta depth limit of 6. Our data makes sense because as the board size increases more nodes need to be expanded. By a lower depth limit it is expected to have more ties. The tradeoff is that it takes less compute time. With heuristic and without heuristic there is a clear difference between the number of nodes expanded (hence more time taken). Thus we decided to keep a consistent, lower depth limit of 6.

Heuristic ordering Given the results from Table 1, heuristic ordering significantly helps speed up the search process by reducing the number of node expanded.