# Report on Final Project

Bach Xuan Phan and Mykyta Synytsia

Abstract

This paper compares the performance of two adversarial search algorithms that are designed to play Reversi, a depth limited minimax with alpha-beta pruning and a Monte Carlo Tree Search with selection policy. The two agents play against each other in a series of games via simulation developed in Python with several combinations of parameters including board size (4, 8, 12), search depth for minimax (3, 4, 5, 6) and iterations for MCTS (10, 100, 1000, 10000). Based on these simulations the effectiveness of the algorithms is evaluated using search times, number of nodes explored, and win rate. Results show that minimax spends less time on the search than MCTS with similar estimated node exploration numbers and achieves a higher win rate, especially notable for a board size of 8.

## 1 Introduction

Problem solving by search is a fundamental principle in artificial intelligence as it isn't always immediately obvious which action is the correct [1]. Fully observable turn-based games with deterministic rules and environments such as Chess or Go are commonly studied in artificial intelligence as these games have a tremendous state space and a very high branching factor, meaning that an exhaustive search is practically impossible due to both space and time complexity. To solve such problems, agents typically utilize adversarial search algorithms to find the optimal action in a state.

One such game is Reversi, a two-player turn-based game with a deterministic set of actions, rules and a fully observable environment. Each player has 64 discs of their color (black or white) the goal of each player is to fill as many cells as possible with their discs within an 8x8 grid. The discs may be placed adjacent to their opponents' discs only if the players disc outflanks one or more of the opponents' discs resulting in a capture. The game is over once the entire grid is filled and the player with most discs on the board wins.

We developed two adversarial search agents that play Reversi, the first utilizes Minimax search and the second uses Monte Carlo Tree Search (MCTS). We are interested in the effect of search depth for minimax and iterations for MCTS on performance of the two algorithms when facing off against one another in a series of games based on metrics such as search time, number of nodes explored and win rate. Additionally, we also test with several board sizes, ranging from 4x4 to 12x12.

## 2 Methodology

To evaluate the effect of search depth on minimax and iterations on MCTS, we developed a simulation in Python that simulates a series of Reversi games where the search agents play against one another with various

combinations of search depth and iterations. The performance of both algorithms is evaluated based on game time, total search time, number of nodes explored, average time thinking per move, maximum thinking time per move, and win rate. These metrics allow us to compare the effectiveness and efficiency of the two algorithms. Python was chosen as the main programming language because of its ease of use, readability and quick prototyping time.

## 2.1 Reversi

Reversi, also known as Othello, is a two-player strategy board game played on an 8x8 grid. The game starts with four discs placed in the center of the board in a square — two white and two black — with players taking turns placing their discs. One player controls the black discs and the other controls the white discs. Black always moves first. The objective is to have the most discs on the board by the end of the game. With an 8x8 grid, the game ends after 60 moves and has a branching factor of approximately 10 resulting in a game tree size of approximately $10^{60}$ making it a challenging problem for AI.

A legal move in Reversi consists of placing a disc on the board such that it brackets one or more of the opponent's discs between the new disc and another of the player's existing discs in a straight line (horizontal, vertical, or diagonal). All bracketed discs are flipped to the player's color. If a player cannot make a legal move, they must pass their turn. The game ends when neither player can move — typically when the board is full or no more flips are possible. Strategic play involves controlling the corners and edges, which provides more stability and influence over the board. The game combines simple rules with deep tactical possibilities, making it easy to learn but challenging to master.

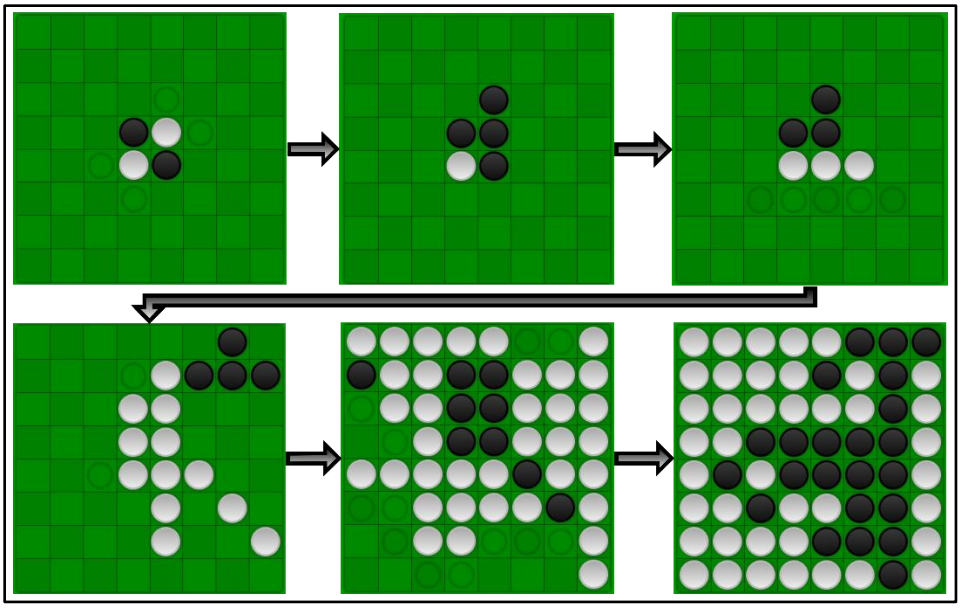

Fig 1. A game of Reversi, where the player with white discs wins

## 2.2 Search algorithms

Subsections 2.2.1 and 2.2.2 describe and summarize the minimax search and MCTS algorithms that were used for the experiment. Minimax utilizes a utility function to determine the best possible move in some state by

performing a search and evaluating terminal nodes. However, in very large state space, performing a search until terminal nodes are reached takes large space and time complexity, so a depth limited minimax is used which makes use of a heuristic evaluation function to determine the utility of a node. On the other hand, MCTS performs numerous iterations of random playouts, playing against itself, starting from some state to determine the most effective move, statistically. One problem of selecting actions randomly is that the algorithm assumes both players play randomly, therefore a selection policy is utilized to dedicate simulations to actions which lead to better outcomes, or to actions which haven't been explored much.

### *2.2.1 Minimax*

The minimax search algorithm can be thought of as recursive depth first search, each node is a min or max node, which correspond to opponent and player, respectively. These nodes alternate with every move. Upon reaching a terminal state the leaf's value is computed based on a utility function and propagated to the root where recursion unwinds. Max aims to maximize the decision to select the best possible decision, while min selects the worst possible outcome for max; this results in assuming that both players will always play optimally. This allows the root node to select the best possible action in any given state.

For vast state spaces, an exhaustive depth first search requires exponential space and time complexity, so a depth-limited search is used. For example, a full game of Reversi may take up to 60 moves and has an average branching factor of 10. As terminal states are not guaranteed to be reached in depth limited minimax, a heuristic evaluation function is required to determine the expected value of a leaf node. For Reversi, we used a weighted grid which has higher values for preferred positions such as corners along with mobility score based on number of moves available. The formula below describes the heuristic evaluation function for node n:

$$score(n) = w_{pos} * \left(\sum n.discs(p) * w_b - \sum n.discs(o) * w_b\right) + w_{mob} * (n.moves(p) - n.moves(o))$$

Where n is the current node and p and o correspond to player and opponent, respectively. $w_{pos}$ and $w_{mob}$ correspond to a positional weight and mobility weight, we used a constant weight of 10 and 5. n.discs(x) returns a matrix where indices are 1 for locations where player x's disks are present, 0 otherwise. $W_b$ is a matrix with weights for each disc position (shown in Figure 2). The number of valid moves in the current state for player x is designated as n.moves(x). With this heuristic evaluation function, scores are greater for the player when they have more discs on the board than the opponent and/or when the number of possible moves is greater.

**4x4**

| | | | |
|---|---|---|---|
| 120 | -20 | -20 | 120 |
| -20 | -40 | -40 | -20 |
| -20 | -40 | -40 | -20 |
| 120 | -20 | -20 | 120 |

**6x6**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 120 | -20 | 20 | 5 | 5 | 20 | -20 | 120 |
| -20 | -40 | -5 | -5 | -5 | -5 | -40 | -20 |
| 20 | -5 | 15 | 3 | 3 | 15 | -5 | 20 |
| 5 | -5 | 3 | 3 | 3 | 3 | -5 | 5 |
| 5 | -5 | 3 | 3 | 3 | 3 | -5 | 5 |
| 20 | -5 | 15 | 3 | 3 | 15 | -5 | 20 |
| -20 | -40 | -5 | -5 | -5 | -5 | -40 | -20 |
| 120 | -20 | 20 | 5 | 5 | 20 | -20 | 120 |

**12x12**

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 120 | -20 | 20 | 5 | 3 | 3 | 3 | 3 | 5 | 20 | -20 | 120 |
| -20 | -40 | -5 | -5 | -3 | -3 | -3 | -3 | -5 | -5 | -40 | -20 |
| 20 | -5 | 15 | 3 | 3 | 3 | 3 | 3 | 3 | 15 | -5 | 20 |
| 5 | -5 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | -5 | 5 |
| 3 | -3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | -3 | 3 |
| 3 | -3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | -3 | 3 |
| 3 | -3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | -3 | 3 |
| 3 | -3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | -3 | 3 |
| 5 | -5 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | -5 | 5 |
| 20 | -5 | 15 | 3 | 3 | 3 | 3 | 3 | 3 | 15 | -5 | 20 |
| -20 | -40 | -5 | -5 | -3 | -3 | -3 | -3 | -5 | -5 | -40 | -20 |
| 120 | -20 | 20 | 5 | 3 | 3 | 3 | 3 | 5 | 20 | -20 | 120 |

Fig 2. Position weights for board sizes of 4x4, 8x8 and 12x12

To further reduce the complexity of the search we also utilize a pruning strategy called alpha-beta pruning. The alpha term corresponds to the highest value that has been encountered, and beta stores the lowest value. These values are propagated to the root node when recursion unwinds to update alpha beta values for the root node which bounds the search tree to between those values. When the utility of some path evaluates to outside the alpha-beta values, it is pruned as there is no point in exploring paths that lead to worse outcomes. The pseudocode is summarized in Function 1.

```
Function minimax(state, depth, alpha, beta, isMaxNode):
    if depth == 0 or state is terminal then
        return state.evaluate()      # returns score of state or a large positive or negative value
                                     # if state is terminal
    moves = state.valid_moves()      # get valid moves for current player

    if moves is empty then           # switch players if no valid move available
        state_copy = copy.state
        state_copy.player = not state.player
        return  minimax(state_copy, depth - 1, alpha, beta, not isMaxNode)

    if isMaxNode then                                # max player
        max_eval = -infinity
        for each move in moves do                    # try each move for max player
            state_copy = copy(state)
            state_copy.make_move(move)
            state_eval = minimax(state_copy, depth - 1, alpha, beta, False)
            max_eval = max(max_eval, state_eval)
            alpha = max(alpha, state_eval)
            if beta <= alpha then break              # beta cutoff, nodes are pruned
        return max_eval
    else                                             # min player
        min_eval = infinity
        for each move in moves do                    # try each move for min player
            state_copy = copy(state)
            state_copy.make_move(move)
            state_eval = minimax(state_copy, depth - 1, alpha, beta, True)
            min_eval = min(min_eval, state_eval)
            beta = min(beta, state_eval)
            if beta <= alpha then break              # alpha cutoff, nodes are pruned
        return min_eval
```

Func 1. Pseudocode for minimax search with alpha beta pruning.

### *2.2.2 Monte Carlo Tree Search*

Monte Carlo Tree Search (MCTS) consists of running simulations starting from the current state of the search until a terminal state is reached. Moves are chosen at random or by utilizing a playout policy for both player and opponent. Typically, these playouts are done over thousands of iterations to determine which action led to the most optimal outcome based on the average utility or win percentage in the case of Reversi. Running MCTS completely at random can be inefficient as it takes many iterations to determine the most optimal outcome. Another issue is that although the optimal action can be found, base MCTS assumed that both players are playing at random.

A playout policy involves selectively dedicating more simulations to important parts of the game tree. This benefits the search in those states with the most potential are simulated more often, and certain paths are ultimately pruned as they lead to less desirable outcomes. Such policies involve keeping track of the average utility or win rate for actions in some state throughout iterations, which favors actions that produce higher utility values. In our case we used upper confidence bounds applied to trees (UCT), which makes use of the upper confidence bound (UCB1) formula, shown below, for ranking all possible moves [1].

$$UCB1(n) = \frac{wins(n)}{total\_games(n)} + C * \sqrt{\frac{\log(\text{visits(n. parent)})}{visits(n)}}$$

UCB1 selection policy for node n, shown in Formula 1, balances exploration (exploring states which were visited less) and exploitation (exploring states which have performed well before) [1]. This helps to evaluate the utility of node n based on exploitation, win rate of node n, and exploration, which is based on how often the node has been visited with respect to the parent. Along comes a balancing constant, C, a popular value for which is the square root of 2. The exploitation factor rises as the win rate increases, and when a node is not explored very much in relation to the parent, the exploration factor increases. This selection policy prioritizes resources to actions which statistically lead to better outcomes. The pseudocode is summarized in Function 2.

```
Function mcts(state, iterations):
    moves = state.valid_moves()                    # get valid moves for current state

    if moves is empty then return None             # skip if no moves are available
    if length(moves) == 1 then return moves[0]     # if only one move is available, return it

    for iteration in iterations do
        node = state

        # use selection policy until a node with untried moves is reached
        while node.untried_moves is empty and node.has_children then        # node selection
            node = select node based on UCB1 formula

        if node.untried_moves is not empty then                             #node expansion
            move = select node from node.untried_moves at random
            node_copy = copy(node)
            node_copy.make_move(move)
            node = node_copy

        board_copy = copy(node.board)
        while board_copy is not terminal do                 # simulation
            valid_moves = board_copy.valid_moves
            if valid_moves is empty then switch player
            move = select random action from valid_moves
            board_copy.make_move()

        while node is not None do                           # backpropagation
            result = 1.0 if player 1 won
            result = 0.5 if draw
            result = 0 if player2 won
            node.visits += 1
            node.wins += result
            node = node.parent

    return move from root with highest win rate
```

Func 2. Pseudocode for MCTS search.

## 2.4 Test design

This section discusses the test design for comparing minimax and MCTS algorithms when used for playing Reversi. We compare the performance of algorithms based on a series of Reversi game simulations where the two algorithms play against one another on several board sizes, minimax search depths and MCTS simulation iterations. For each combination of parameters 6 Reversi games are played by the two algorithms, for half the games minimax makes the first move and MCTS makes first move for the other half. The parameter combinations for every six games are summarized below:

- Board sizes of 4x4, 8x8, 12x12
- Minimax search depths of 3, 4, 5 and 6
- MCTS simulation iterations of 10, 100, 1000 and 10000

These combinations of parameters give us 6*3*4*4 = 288 total games. Using the results of these games, we compare the effectiveness of the search algorithms by considering metrics such as search time, nodes explored and win rate. Additionally, we limit searches to a maximum time duration of 30 minutes; searches which exceed this duration are considered failures. The algorithm which exceeded the time limit first was considered the loser.

Our decision to use these search depths and MCTS iteration has to do with calculations regarding approximate upper bound of nodes explored for a board size of 8. Since the average branching factor per move is 10, an approximation of search space can be computed for various search depths and MCTS iterations. For minimax, the approximate upper bound on the number of nodes explored per move is $10^d$, where d is depth. For MCTS, since each game is less than or equal to 60 moves an approximate number of nodes explored per move is 60i, where i is the number of iterations. We chose search depths and iteration counts which result in a similar approximate number of nodes explored per move.

# 3 Result

This section presents the results in the form of tables, plots and summaries of observations based on search time, total and average nodes explored and win rate for both algorithms. Note that searches which surpassed a duration of 30 minutes did not play out; therefore, results for games where searches failed are inaccurate.

## 3.1 Search time

In this subsection, the average search time and average time per move are presented in Table 1 and 2 for minimax, and Table 3 and 4 for MCTS. These results show the effect of larger board sizes, search depths, and iterations on the duration of the searches. For minimax, Tables 1 and 2, the search depth has a much more dramatic effect on the search duration for board sizes of 8 and 12 rather than a board size of 4. For example for board size of 8 and depth of 3 the average total search time is ~6.8 seconds, and for a depth of 6, the search time is ~632.8 seconds.

For MCTS, Tables 3 and 4, the effect of iterations is much more noticeable than in minimax even at board sizes of 4. With a board size of 4 and 10 iterations, the search time was ~44 milliseconds, and over 12 seconds for 10000 iterations. The effect is even more noticeable at board sizes of 8, where 10 iterations only took ~4 seconds and over 30 minutes with 10000 iterations. This shows one of the drawbacks of MCTS which typically simulates hundreds or thousands of playouts of the entire game, in this case, to determine which action leads to most winning outcomes. Running thousands of simulations of entire playouts can be time consuming, even if moves are chosen at random.

| **Minimax average of total search time (s) vs board size and search depth** | | | | |
|---|---|---|---|---|
| **Board Size** | **Depth** | | | |
| | **3** | **4** | **5** | **6** |

| | | | | |
|---|---|---|---|---|
| **4** | 0.022 | 0.04 | 0.077 | 0.121 |
| **8** | 6.808 | 30.616 | 167.335 | 632.757 |
| **12** | 87.183 | 390.696 | 856.841 | 1376.846 |

Table 1. Average of total search time for minimax.

| **Minimax average time per move (s) vs board size and search depth** | | | | |
|---|---|---|---|---|
| **Board Size** | **Depth** | | | |
| | **3** | **4** | **5** | **6** |
| **4** | 0.004 | 0.007 | 0.014 | 0.02 |
| **8** | 0.283 | 1.208 | 6.984 | 23.356 |
| **12** | 1.535 | 6.786 | 37.227 | 92.765 |

Table 2. Average time per move for minimax.

| **MCTS average of total search time (s) vs board size and iterations** | | | | |
|---|---|---|---|---|
| **Board Size** | **Iterations** | | | |
| | **10** | **100** | **1000** | **10000** |
| **4** | 0.044 | 0.299 | 1.834 | 12.024 |
| **8** | 4.062 | 39.075 | 370.572 | 1888.147 |
| **12** | 34.755 | 373.311 | 1772.112 | 2851.605 |

Table 3. Average of total search time for MCTS.

| **MCTS average time per move (s) vs board size and iterations** | | | | |
|---|---|---|---|---|
| **Board Size** | **Iterations** | | | |
| | **10** | **100** | **1000** | **10000** |
| **4** | 0.007 | 0.049 | 0.305 | 1.985 |
| **8** | 0.14 | 1.299 | 12.235 | 226.017 |
| **12** | 1.122 | 9.925 | 143.197 | 1428.281 |

Table 4. Average time per move for MCTS.

## 3.2 Nodes explored

In this section, the average total number of nodes and average number of nodes explored per move is shown in Table 5 and 6 for minimax, and Table 7 and 8 for MCTS. These results highlight the effect of board size, depth and iterations on the search space. Note that these results are for entire games, not singular moves, apart from Table 6 and 8 which averages nodes explored over number of moves. With minimax and a board size of 4, the number of nodes explored nearly doubles at each increment depth. Focusing a board size of 8, at a depth of 3, the number of nodes explored is nearly 7000 and nearly 7.5 million for a depth of 6 (Table 5).

However, MCTS is not any better as the number of nodes explored, even for board sizes of 4, increases significantly. This ties into the results from section 3.1, which presents the effect of iterations on search time. From Table 5, we see that with a board size of 4 and 10 iterations, over 300 nodes were explored, but at 10000 iterations nearly 60000 nodes were explored by MCTS. For board sizes of 8 and 10 iterations nearly 9000 nodes were explored and over 4 million with 10000 iterations.

| **Minimax average nodes explored vs board size and search depth** | | | | |
|---|---|---|---|---|
| **Board Size** | **Depth** | | | |
| | **3** | **4** | **5** | **6** |
| **4** | 104.04 | 196 | 381.5 | 658.42 |
| **8** | 6924.04 | 32816.58 | 180567.2 | 745303 |
| **12** | 41911.46 | 189724.4 | 319085.7 | 666029 |

Table 5. Average nodes explored for minimax.

| **Minimax average nodes explored per move vs board size and search depth** | | | | |
|---|---|---|---|---|
| **Board Size** | **Depth** | | | |
| | **3** | **4** | **5** | **6** |
| **4** | 18.73 | 33.27 | 69.41 | 111.44 |
| **8** | 275.82 | 1248.26 | 7186.5 | 27659.54 |
| **12** | 688.14 | 3093.52 | 13476.16 | 43797.56 |

Table 6. Average nodes explored per move for minimax.

| **MCTS average nodes explored vs board size and iterations** | | | | |
|---|---|---|---|---|
| **Board Size** | **Iterations** | | | |
| | **10** | **100** | **1000** | **10000** |
| **4** | 321.67 | 2237.54 | 12231.88 | 59256.71 |
| **8** | 8959.71 | 87142.33 | 836543.3 | 4195134 |
| **12** | 32708.46 | 361819.5 | 1598819 | 2729862 |

Table 5. Average nodes explored for MCTS.

| **MCTS average nodes explored per move vs board size and iterations** | | | | |
|---|---|---|---|---|
| **Board Size** | **Iterations** | | | |
| | **10** | **100** | **1000** | **10000** |
| **4** | 54.47 | 368.21 | 2031.54 | 9769.24 |
| **8** | 308.2 | 2897.44 | 27625.41 | 470541.5 |
| **12** | 1014.28 | 9390.97 | 124830.6 | 1337428 |

Table 8. Average nodes explored per move for MCTS.

## 3.3 Win rate

The section summarizes results regarding win rate for minimax in Table 9 with different board sizes, search depths and MCTS iterations. It is important to note that win rates where searches were stopped early aren't exactly accurate as the algorithm which exceeded the 30-minute limit was considered the loser which primarily occurred with board size of 12, see 3.4. Focusing primarily on the average win rate of minimax in Table 9, MCTS significantly outperforms minimax in terms of win rate for board size of 4.

Something that wasn't expected is that a depth of 3 achieves a higher win rate than other depths for a board size of 4, regardless of the MCTS iterations, where the average win rate was ~41% for minimax. However, with larger board sizes, 8 and 12, minimax generally achieves a higher win rate. With a board size of 8 and depths of 3, 4, 5 and 6 minimax's average win rate was approximately 42, 63, 54 and 71 percent, respectively.

| Minimax win rate vs board size | | | | | | |
|---|---|---|---|---|---|---|
| **Board Size** | **Depth** | **Iterations** | | | | **Average** |
| | | **10** | **100** | **1000** | **10000** | |
| **4** | **3** | 33.33% | 33.33% | 50.00% | 50.00% | 41.67% |
| | **4** | 16.67% | 0.00% | 0.00% | 0.00% | 4.17% |
| | **5** | 16.67% | 0.00% | 0.00% | 0.00% | 4.17% |
| | **6** | 16.67% | 0.00% | 0.00% | 0.00% | 4.17% |
| **8** | **3** | 100.00% | 0.00% | 16.67% | 50.00% | 41.67% |
| | **4** | 100.00% | 66.67% | 16.67% | 66.67% | 62.50% |
| | **5** | 100.00% | 50.00% | 16.67% | 50.00% | 54.17% |
| | **6** | 100.00% | 33.33% | 66.67% | 83.33% | 70.83% |
| **12** | **3** | 83.33% | 66.67% | 100.00% | 100.00% | 87.50% |
| | **4** | 83.33% | 33.33% | 100.00% | 83.33% | 75.00% |
| | **5** | 100.00% | 100.00% | 100.00% | 83.33% | 95.83% |
| | **6** | 100.00% | 83.33% | 83.33% | 66.67% | 83.33% |

Table 9. Win rate for minimax.

## 3.4 Failed searches

This section presents results regarding failed searches due to exceeding the time limit in Table 10. This table shows that for board sizes of 4, none of the searches exceeded the time limit. For board sizes of 8 and 12, the searches which exceeded the time limit were with MCTS iterations of 10k. Additionally for board size of 12, the only depth and iteration combinations which completed all games were with minimax depths of 3 and 4, and MCTS iterations of 10 and 100, in remaining parameter combinations there was at least 4 searches that exceeded the time limit.

| Games stopped early vs board size, depth and MCTS iterations | | | | | |
|---|---|---|---|---|---|
| **Board Size** | **Depth** | **Iterations** | | | |
| | | **10** | **100** | **1000** | **10000** |
| **4** | **3** | 0 | 0 | 0 | 0 |

| | | | | | |
|---|---|---|---|---|---|
| | **4** | 0 | 0 | 0 | 0 |
| | **5** | 0 | 0 | 0 | 0 |
| | **6** | 0 | 0 | 0 | 0 |
| **8** | **3** | 0 | 0 | 0 | 6 |
| | **4** | 0 | 0 | 0 | 6 |
| | **5** | 0 | 0 | 0 | 6 |
| | **6** | 0 | 0 | 0 | 6 |
| **12** | **3** | 0 | 0 | 6 | 6 |
| | **4** | 0 | 0 | 5 | 6 |
| | **5** | 4 | 5 | 5 | 6 |
| | **6** | 6 | 5 | 6 | 6 |

Table 10. Failed searches due to exceeding time limit.

# 4 Discussion

## 4.1 Effect of board size on algorithms

Based on the results presented in sections 3.1 and 3.2, we can see that board size has a drastic effect on the algorithms which is shown both by total search time (see section 3.1) and number of nodes explored (see section 3.2). This is due to exponentially larger game tree sizes with increasing board sizes. As mentioned in 3.1, for minimax we saw that with smaller game tree sizes the difference in duration of the searches was quite insignificant for board sizes of 4, but much more dramatic for board sizes of 8 and 12.

It was also interesting to see that minimax performed worse in terms of win rate with a board size of 4, regardless of depth (Table 9). In preliminary testing we thought that minimax spent more time on the search, however we see in Table 1 and Table 3 that MCTS spent a much longer time on the searches. This is also shown by the total number of nodes explored, MCTS generally explored many much more nodes than minimax (see section 3.2). This is due to the nature of the MCTS algorithm which simulates entire playouts of the game for over, typically thousands, of iterations.

## 4.2 Effect of minimax depth

From results in sections 3.1, 3.2 and 3.3 we see that the depth of minimax makes a big difference on the duration of the search, number of nodes explored and win rate but as mentioned before it only seems to do better with larger state spaces which is indicated by the low win rate of minimax regardless of depth for a board size of 4. However, minimax generally outperforms MCTS for board size of 8 and 12 (see section 3.3). We also see that the number of nodes explored increases considerably with board size which is expected as the branching factor and total number of moves increases with board size.

### 4.3 Effect of MCTS iterations

MCTS iterations significantly increase both search time and search space which is shown by results in section 3.1 and 3.2. This is expected as MCTS performs entire game simulations from any given state to determine the best action to take statistically. We see in Table 3 and Table 6 how MCTS average search duration and number of nodes explored, respectively, increases significantly with more iterations. This shows the inefficiency of MCTS when used to play out entire games per iteration. Therefore it may make more sense to only go to a certain depth with MCTS in order to reduce search duration and number of nodes explored, but for this to work well a good heuristic/evaluation function must be used.

### 4.4 In general, which algorithm performs better?

To answer this question, we want to evaluate the performance of the algorithms based multiple metrics including search duration, number of nodes explored and win rate. However, we will focus only on board size of 8 as this is the typical board size for Reversi. With this in mind, we can see that minimax generally outperforms MCTS at all depths with equivalent node exploration numbers. This indicates that for Reversi, minimax might be the better algorithm. However, it is hard to say without playing against these algorithms. So, we propose a future experiment to answer this question better, which would evaluate the difficulty playing against these algorithms based on input from an expert Reversi player.

## 5 Conclusion

This study compared the performance of two adversarial agents that utilize a depth limited minimax with alpha-beta pruning and MCTS with selection policy for playing a two-player turn-based game, Reversi. Where the goal is to have the greatest number of discs on the board at the end of the game by outflanking opponent pieces. Minimax algorithm performs a depth first search and backpropagates the utility of the leaf node to select the optimal action at the root state. On the other hand, MCTS performs a series of simulations until a terminal state is reached to select the action that produces the best outcome probabilistically.

To evaluate and compare the two agents, 288 total games of Reversi are played where the agents face off against one another at varying board sizes, search depths for minimax and simulation iterations for MCTS. These games are evaluated based on various metrics including total time, time per move, number of nodes explored and win rate to highlight important trade-offs between the two algorithms. Based on the results in section 3, we see that minimax generally takes less time than MCTS and performs better overall for a board size of 8, which is the default for Reversi games, this result was not what we expected since MCTS is preferred for larger game trees in order to make the search more efficient but it turns out that it doesn’t always find the optimal decision which is indicated by the win rate of minimax with various combinations of search depths and MCTS iterations.

## References

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Hoboken, NJ, USA: Pearson, 2021.