

# DAT410: Group 60 Assignment 4

**Cecilia Nyberg**

**Elin Stiebe**

Personal number: 990106

Personal number: 000210

Program: MPDSC

Program: MPDSC

Hours spent: 34

Hours spent: 34

`cecnyb@chalmers.se`

`elinsti@chalmers.se`

February 27, 2024

*We hereby declare that we have both actively participated in solving every exercise. All solutions are entirely our own work, without having taken part in other solutions.*

## 1 Reading and reflection

*Read and summarise (briefly) the paper Mastering the game of Go with neural networks and tree search*

“Mastering the game of Go with deep neural networks and tree search”, 2016 describes that the game Go was an extra challenging game to model in ML since has has high complexity with its 150 legal moves (its breadth) and long game duration (its depth). Consequently, the entire search space cannot be examined, it must be shrunk instead. The breadth is reduced by *position value*,  $v^*(s)$  and its depth by a sampling of actions from a policy  $p(a/s)$ . “Mastering the game of Go with deep neural networks and tree search”, 2016 searched the defined search space with *Monte Carlo Tree Search*, *MCTS* which was extended with policies from humans playing Go. The human actions told them how to sample  $p(a|s)$ .

“Mastering the game of Go with deep neural networks and tree search”, 2016 successfully created an algorithm that combines MCTS with value and policy networks. Their Go system performed stronger than previous programs winning 494 out of 495 games played against them (winning 99.8% of the games). They structured the algorithm in the following way:

1. They passed the board positions as an image and used a CNN layer to turn it into representations of positions
2. Trained a supervised learning policy network on human professional moves.

3. Trained a fast policy that can sample actions.
4. Trained a reinforcement learning layer that improves on the SL network of human professional knowledge.
5. Trained a value network to predict the winner as the RL layer played against itself.

As mentioned the team in “Mastering the game of Go with deep neural networks and tree search”, 2016 used an MCTS algorithm combined with policy and value networks. The MCTS algorithm works by traversing the tree and updating the valuations of different moves to then make the move that is most valuable for the player. Each edge in the tree stores an action value  $Q(s, a)$  and a visit count  $N(s, a)$ . The actions are chosen to maximize the action value plus a "bonus". “Mastering the game of Go with deep neural networks and tree search”, 2016 define the equation as,

$$a_t = \arg \max_a (Q(s_t, a) + u(s_t, a)) \quad (1)$$

The bonus term is defined as,

$$u(s, a) \propto \frac{P(s, a)}{1 + N(s, a)} \quad (2)$$

One fact they had to consider the effects of is that though larger networks play better it also takes them for them to evaluate moves. This can be negative if there is a time limit on how long the players can think. Another aspect is overfitting. In the case of “Mastering the game of Go with deep neural networks and tree search”, 2016 their policy network avoids overfitting by not only playing according to its policy but also randomly picking samples from a pool of opponent policy.

## 2 Implementation

*Implement tree search (Monte-Carlo or otherwise) in order to learn to win (or at least not lose) at the game Tic-tac-toe on the classic 3x3 grid. You will have to make choices about the roll-out policy of your player the policy of your opponent the selection policy in your search tree the updates to your search tree ("back-up") Describe and motivate these choices briefly. For example, if you use a non-Monte-Carlo search, what did you use? If you use MC, how did you implement sampling? Evaluate your algorithm and comment on its pros and cons. For example, is it fast? Is it sample efficient? Is the learned policy competitive? Does it lose? Would you, as a human, beat it? Would it scale well to larger grids such as 4x4 or 5x5?*

The implementation of the tic-tac-toe consists of a board class, a node class, a machinePlayer class, and a class for the tic-tac-toe game. The node class holds a reference to a board, which can be viewed as a state. A node also holds references to its children and its parent.

There were many decisions to make when implementing the game that all affected the performance of the machine player. The Monte Carlo tree search happens in the machine class and is divided into four parts; selection, expansion, simulation, and backpropagation. The following sub-sections dive deeper into each of the four steps.

### 2.1 Selection

In the selection method, it is decided which end node, *leaf*, should be expanded. The leaf is chosen by calculating the UCT score and choosing the leaf with the highest one.

The search for the node with the highest UCT score is done by starting at the root and choosing the child with the highest UCT until an end node is reached. The implementation of the selection method is shown in listing 1.

```

def selection(self, node):
    if node.is_terminal():
        return node
    if not node.has_children():
        return node

    children = node.get_children()
    # Recursively call selection on the child with max UCB
    max_ucb_child = max(children, key=lambda n: n.get_UCB())
    return self.selection(max_ucb_child)

```

Listing 1: Selection

The UCT score is calculated as shown in equation 3. The  $N_{total}$  is the cumulative sum of visits of nodes. The  $n_i$  is the number of visits to node  $i$ . Since division by zero is not possible, if  $n$  is zero the UCB score is set to infinity. This means that if there exists an end node that has not been visited it will always be chosen. The  $2$  in the second term is the *exploration weight*, if the weight is given a higher value exploration is prioritized over-exploitation, conversely, exploitation is prioritized with a lower weight.

$$\frac{Q}{n_i} + 2 \cdot \sqrt{\frac{\log(N_{total})}{n_i}} \quad (3)$$

The implementation for the UCT score calculation is shown in listing 2.

```

def get_UCB(self):
    total_N = self.get_total_N()
    if self.n == 0:
        # Inf means it has not been visited yet
        return float('inf')
    if total_N < 0:
        print('the total N is', total_N, 'for node', self.state.get_board())
    return self.Q/self.n + 2 * (math.log(total_N)/ (self.n)) ** 0.5

def get_total_N(self):
    if self.parent is None:
        # Base case: if there's no parent, return the N value (root reached)
        return self.n
    else:
        # Recursive case: add current node's N to parent's total
        return self.parent.get_total_N()

```

Listing 2: Get UCB score

## 2.2 Expansion

In the expansion method, the chosen end node is extended with all possible child nodes. Each child node reflects a possible action to take from the chosen end node. A random one of these children is then returned from the expansion method. The implementation for the expansion is shown in listing 3.

```

def expansion(self, node):
    if node.is_terminal():
        return node

    board = node.state.create_copy()
    possible_moves = board.get_possible_moves()
    for move in possible_moves:
        new_board = board.create_copy()
        new_board.make_move(move[0], move[1])
        new_node = Node(new_board)
        new_node.set_parent(node)
        node.add_child(new_node)


    #pick a random child
    if len(node.get_children()) > 0:
        random_child = random.choice(node.get_children())
        return random_child
    return node

```

Listing 3: Expansion

## 2.3 Simulation

The randomly chosen child node is the state where the simulation will start. It is therefore sent to the simulation method. The simulation method works by iteratively choosing a possible action from the current state and executing this action, thereby moving to a new state. This goes on until the simulated game is over, either because someone won or the board is full.

The action taken is decided by iterating through the possible moves from the current state and checking firstly if any of these moves lead to a win, and secondly, if a move blocks the opponent from winning. If a win is found, that action is chosen and executed. Otherwise, if a block is found, that action is performed to stop the opponent from winning. This priority is done because if a win is possible it doesn't matter if the opponent has a chance of winning the next time. But if no win is possible it does matter and the opponent should be blocked. If neither a win nor a block is possible, a random action is chosen. The same strategy is used for both the machine's turns and the opponent's turns in the simulation. This is done until the game is over. If the game ended with a draw, 0 will be returned, otherwise the sign for the winner will be returned. The implementation is shown in listing 4. 

```

def simulation(self, node):
    if len(node.get_children()) == 0:
        return node.get_state().get_winner()
    else:
        to_sim_from = node.get_state().create_copy()

        while not to_sim_from.game_is_over():
            player = to_sim_from.turn
            possible_moves = to_sim_from.get_possible_moves()
            has_made_move = False

            for possible_move in possible_moves:
                block_move = None
                board_copy = to_sim_from.create_copy()
                if board_copy.move_block_opponent(possible_move[0],
                    possible_move[1]):
                    block_move = possible_move

                board_copy.make_move(possible_move[0], possible_move[1])
                if board_copy.get_winner() == player:
                    to_sim_from.make_move(possible_move[0], possible_move[1])
                    return to_sim_from.get_winner()

            if block_move is not None:
                to_sim_from.make_move(block_move[0], block_move[1])
                has_made_move = True

            if not has_made_move:
                move = random.choice(possible_moves)
                to_sim_from.make_move(move[0], move[1])

        return to_sim_from.get_winner()

```

Listing 4: Method for simulation

## 2.4 Backpropagation

After the simulation, backpropagation is executed to update the  $Q$  and  $v$  values for the parent nodes of the explored node. If the simulation from the explored node results in a win for the machine, the  $Q$  and  $n_i$  values for this node and all following parent nodes are updated by adding 1 to them. This means that the  $n_0$  node, the  $n$  value for the root node, is updated by adding 1 every time a simulation has been performed. Therefore it holds the value for  $N_{total}$ . The code for the backpropagation is shown in listing 5.

```
def backpropagate(self, node, result):
    current_node = node
    while current_node is not None:
        current_node.n += 1
        if result == self.player:
            current_node.Q += 1
        elif result == 0:
            current_node.Q += 0
        else:
            current_node.Q -= 1
        current_node = current_node.parent
```

Listing 5: Method for backpropagation.

## 2.5 Machine player

Putting these four methods together gives the MCTS algorithm. Though, only using MCTS had a negative consequence, immediate wins or losses can be missed. If there is an immediate possibility to win or block the opponent from winning in the next move, this is of course the best thing to do when playing tic-tac-toe. It was deemed unnecessary to perform the whole MCTS algorithm if an immediate win was possible, therefore this was checked before the MCTS was started. The implementation can be seen in listing 6.



```

def monte_carlo_tree_search(self, board):
    root = Node(board)

    if self.immediate_win_or_block(root) is not None:
        return self.immediate_win_or_block(root)

    for i in range(self.max_depth):
        leaf = self.selection(root)
        leaf = self.expansion(leaf)
        simulation_result = self.simulation(leaf)
        self.backpropagate(leaf, simulation_result)

    return self.get_move_made(self.get_best_child(root).get_state(),
                              root.get_state())

```

Listing 6: MCTS

## 2.6 Discussion

This subsection takes a closer look at the implementation and discusses the choices made throughout.

### 2.6.1 Roll-out Policy for MachinePlayer and Opponent

The roll-out policy is set in the simulation. At first, an implementation was made that simulated random moves and with that sometimes missed important actions such as those leading to a direct win or lose. The roll-out policy was altered to choose the action that led to an immediate win if it existed, or otherwise choose the action that blocked the opponent from having the chance of an immediate win in the next move. If neither of these existed a random action was made.

Now to the opponent. At first, the opponent had a random strategy. This led the model to have a very bad performance, missing obvious strategic choices leading to a direct win for the opponent. The opponent's simulated strategy was modified to have the same strategy as the machine, meaning prioritizing moves that led to their win or secondly, blocking the opponent.

This roll-out policy for the machine and its opponent in the simulation led to a great improvement in the performance of the model since it could find good routes down the tree much faster when not just picking routes completely at random. On the other hand, it leads to increased complexity. The first implementation where each action in the simulation was chosen randomly had a worst-case complexity of  $O(M)$ , where  $M$  is the maximum number of iterations made until the game was over.

This is equal to the maximum number of actions (for this 3\*3 game it is 9) because after that the board is full and the game will terminate. The reason for the complexity being  $O(M)$  is because picking a random action has a constant time complexity ( $O(1)$ ). When the implementation was updated to instead look through all possible actions, the complexity became bigger. If  $M$  is the number of actions, the worst-case complexity for the while-loop looking through all possible actions becomes:

$$O(M + (M - 1) + (M - 2) + \dots + 1) = O\left(\frac{M(M + 1)}{2}\right) \approx O(M^2) \quad (4)$$

This worst-case complexity would be received if no wins were found and the simulated game went on until the board was full. The check winner method called in the while loop leads to even higher complexity for the simulation method. This method has a time complexity of  $O(M)$  since it checks every square on the board and the number of squares is equal to the maximum number of actions. Since this method is called inside the for loop over all possible actions, the time complexity for the simulation becomes:



$$O(M^2 * M) = O(M^3) \quad (5)$$

In this small-scale game, the high complexity doesn't matter that much since the highest possible number of actions  $M$  is 9. However, if it was scaled and used for a larger game, the complexity would become a problem. This is a trade-off between performance and efficiency.

Another important aspect regarding scaling is the storing of states. The storing of states during simulation can be a problem since there can be a large number of states. In the simulation method, each new simulation will create a new instance of the board which is not necessary and instead of doing this, a check potential winner method could be used that checks if a move would lead to a win instead of doing the move first. However, since all references to the old board copy disappear when a new iteration in the while loop starts, Python's garbage collection will delete the old instance of the board. Additionally, instead of creating nodes in the simulation, a copy of the board for the starting node is updated in each iteration. That way, there is no need for backpropagation in the simulation. Instead, the result is updated directly in the starting node and backpropagation is done from there.

### 2.6.2 Selection and expansion policy in the search tree

An important part of the storing of nodes happens in the expansion method. In this implementation, for a chosen end node, all possible children are added to it. This creates a bigger tree than necessary. If instead only one child was added each time the tree would probably not become as large. Since

the UCT score is set to infinity for unexplored nodes, if there exist unexplored nodes in a level of the tree where children are compared, the unexplored node will always be chosen. This leads to a high number of explored states. The formula for the UCT score also has a great impact on the number of explored states, since it can favor exploration or exploitation more or less. The UCT should balance both of these aspects, but by setting the exploration constant,  $c$ , to a higher or lower value, this can be controlled. This is important since not having enough exploration can lead to potentially beneficial actions being missed, and not enough exploitation can result in good actions not being favored and past information not being taken advantage of. High exploitation also leads to more storage usage since more states need to be stored. Once again, if the game was scaled this could become a problem, especially when all possible children are added to an end node. This is also a trade-off, but in this case between performance and memory efficiency.

A related aspect to this is the choice of the  $Q$  value given in the backpropagation. As of now, it is increased with one if the simulation resulted in a win, decreased with one if it resulted in a loss, and untouched if it was a draw. This could have been done differently. For example, information about how far away from the starting node the win was could be taken into consideration. If ten moves had to be made before the machine won, the quality of that node could be lower than if it was only two moves away. Additionally, a draw could also be seen as a loss and result in a decrease in the  $Q$  value. This could result in the algorithm being less prone to re-visit routes that lead to a draw and potentially focus more on routes that lead to a sure win. On the other hand, a draw still results in a lower UCB score because the  $v_i$  value becomes smaller. It could also be possible to have a higher reward for a win and a higher decrease for a loss. This would also make the model more prone to re-visit routes that lead to a win since it gives the exploitation part of the UCB score more impact. It is also possible to take another approach and view a draw as positive since it doesn't lead to the opponent winning. In that case, a draw could have been given a higher  $Q$  value than 0.

In summary, the design of the UCT score affects how the model prioritizes exploration and exploitation, which in turn affects storage usage and also performance. If there is a storage shortage, exploration might need to be down-prioritized. For good performance, a balance between exploration and exploitation is important.

### 2.6.3 Evaluation of Algorithm

Before evaluation of the algorithm, a short note on the `max_depth` parameter in the MCTS algorithm. Setting a high depth such as 1000 would most likely lead to a higher performance, though it would also be slower. In the cases below, `max_depth` was set to 80 to avoid getting a too-slow algorithm but still allow for a rigorous search.

So, how well was the algorithm's performance? The best way to evaluate the performance would

be to get human professionals to try and win against it, as in “Mastering the game of Go with deep neural networks and tree search”, 2016. The performance could then be seen as a percentage of the wins the algorithm would have. Since there was a lack of professional tic-tac-toe players at hand other evaluation options were tried instead. First off, several games against the algorithm were played where some were won by it, but far from every game; A majority of the games were won by humans. Secondly, a dummy game was set up where the algorithm played against a player who time and time again picked random actions. Table 4 demonstrates the results. The algorithm won 958 out of 1000 rounds, resulting in a win frequency of 95.8%. This says something about its performance, it is at least superior to a random player. There were more draws than losses for the machine which was preferred and consequently coded for in the simulation with how Q was updated (+1 for a win, -1 for a loss, and 0 for a tie). The full implementation of the tic-tac-toe match is in listing 7.

Table 1: Game Results: Random acting opponent.

Outcome	Count
Machine wins	958
Random wins	7
Draws	35

```

machine_winns = 0
random_wins = 0
draw=0
total_reps = 1000
for i in range(0, total_reps):
    machine = MachinePlayer('X', 80)
    board = Board(3)
    while not board.game_is_over():
        if board.turn == 'X':
            move = machine.get_best_move(board)
            print("Machine move: ", move)
            board.make_move(move[0], move[1])
            print(board.print_board())
        else:
            moves = board.get_possible_moves()
            move = random.choice(moves)
            print("Random move: ", move)
            board.make_move(move[0], move[1])
            print(board.print_board())
    if board.get_winner() == 'X':
        machine_winns += 1
    elif board.get_winner() == 'O':
        random_wins += 1
    else:
        draw +=1

```

Listing 7: Algorithm vs random player.

Third, another evaluation was made on the algorithm where it meets an opponent that always blocks their wins and always tries to complete its two in a row with a third one. If there is no win or way to block they will play a random move. The full results are in table 2. The machine performs somewhat worse than against the random player getting a win rate of 94.4%. This is reasonable as this opponent has more of a strategy than a random player. Further, most non-wins are still draws, here it becomes clear that the built algorithm favors draws over losses. The altered opponent implementation is in listing 8.

Table 2: Game Results: Winning-Blocking opponent.

Outcome	Count
Machine wins	944
Block-win strategy wins	13
Draws	43

```

moves = board.get_possible_moves()
move = random.choice(moves)
print("Random move: ", move)
board.make_move(move[0], move[1])
print(board.print_board())

```

Listing 8: Algorithm vs Winning-Blocking opponent.

Fourth, one strategy used by players is to get a hold of the middle tile. Consequently, a version of the opponent above was also tried where the middle tile was prioritized before any random tile was put. This strategy turned out to be marginally harder for the algorithm as it got a win rate of 93.7% which is clear in table 3. The coded opponent is in listing 9.

Table 3: Game Results: Winning-blocking-middle opponent.

Outcome	Count
Machine1 wins	937
Win-block-middle strategy wins	0
Draws	63

```

if one_from_win(board) is not None:
    move = one_from_win(board)
    print("Blocking/Winning move: ", move)
    board.make_move(move[0], move[1])
    print(board.print_board())
else:
    if board.get_tile(1, 1) == 0:
        board.make_move(1, 1)
    else:
        moves = board.get_possible_moves()
        move = random.choice(moves)
        print("Random move: ", move)
        board.make_move(move[0], move[1])
    print(board.print_board())

```

Listing 9: Algorithm vs Winning-blocking-middle opponent.

Fifth, letting the machine play against itself most often resulted in a draw Though machine1 won about 11 times as often as machine2. This indicates that there is an advantage in getting to put the first tile.

Table 4: Game Results: Algorithm vs Algorithm.

Outcome	Count
Machine1 wins	640
Machine2 wins	57
Draws	303

Each of the evaluations was run a second time letting the machine play the second move due to the implied advantage above. The results are in table 5. The win rate decreased significantly from the evaluations above, from being in the 90s to being in the 70s.

Table 5: Game Results: All tests above but letting the machine do the second play.

Outcome	Count	Outcome	Count
Machine wins	753	Machine wins	711
Random wins	56	Win-Block strategy wins	124
Draws	191	Draws	165
Outcome		Count	
Machine wins		721	
Win-block-middle strategy wins		31	
Draws		248	

The overall win frequency can be calculated as the average between the two win rates. The machine consequently has the following win rates for each opponent in table 6. An alternative way to evaluate the algorithm against each opponent would be to calculate scores by summing up all wins as +1, losses as -1, and draws as neutrals, this would be another way of evaluating the algorithm performance.

Table 6: Game Results: Win Frequencies Against Each Opponent.

Opponent	Win frequency
Random	85.55%
Winning-Blocking	82.75%
Winning-Blocking-Middle	84.7%

Looking at the evaluations above, the algorithm does behave competitively, it wins a majority of the time and also favors draws before losses, which is a pro of the algorithm. A con related to the evaluation of the different opponents is that the model does not adapt to the specific player it meets but only has one strategy in mind. This in itself can lead to overfitting, therefor it could have been mitigated by doing as “Mastering the game of Go with deep neural networks and tree search”, 2016 did when they mixed their strategy with a sampling of a random choice of actions.

Now, how well does this algorithm scale? The code in listing 10 was run where the algorithm plays against a random player. The random player takes little time, it is the algorithm that takes an increasing amount of time with the increased size of the board.



```

moves = board.get_possible_moves()
move = random.choice(moves)
print("Random move: ", move)
board.make_move(move[0], move[1])
print(board.print_board())

```

Listing 10: Algorithm vs random player.

Table 7 shows how long the match for each size of the table took to run.

Table 7: Game Results: Algorithm vs Algorithm.

Table Size	Time (Milliseconds)
3x3	82.42
4x4	225.75
5x5	638.84
6x6	1658.15
7x7	3070.15
8x8	13481.61
9x9	31937.80

The complexity rises quite drastically. This becomes even more evident when the plot of the times is made. Figure 1 visualizes the results in table 7 and makes it clear that the algorithm scales very badly.

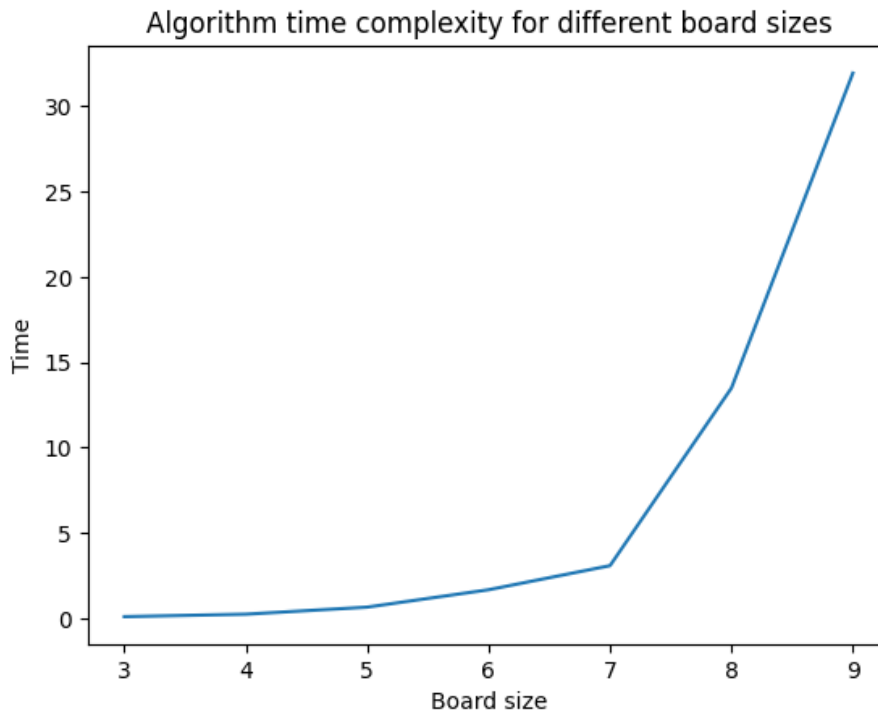


Figure 1: Time of a round as a function of board size.

## 2.7 Characteristics of Tic-Tac-Toe

Tic-tac-toe can be played in many variants shifting the size of the board and the allowed number of pawns out on the field. However, tic-tac-toe is not especially complex. As becomes clear in this assignment, just being the player doing the first move gives you a significant advantage. Since this assignment focused on a 3x3 board, it could have been more efficient to simply implement a rule-based system considering state-action pairs (as discussed, there are not that many to choose from).

Looking back at the summary of “Mastering the game of Go with deep neural networks and tree search”, 2016 in section 1 it becomes clear their algorithm is way superior to the implementation built in this assignment. They managed to reach a winning share of 99.5% which this algorithm is no way near considering how easy tic-tac-toe is to go.



## References

Mastering the game of go with deep neural networks and tree search [Accessed on February 26, 2024]. (2016, January).