Programming an Adaptive Artificial Intelligence using the Monte Carlo Tree Search Algorithm for Turn-Based Games

Table of Contents

Introduction..2

Goal/Objective 4

Methods...4

    Gameplay Framework......................................4

    Monte Carlo Tree Search 6

    Testing...11

Results..12

Discussion 15

    Analysis..15

    Future Implications 16

Conclusion 16

References..17

GameBoard.java (Code).......................................18

Node.java (Code) 24

Introduction

Since the advent of computers, it was soon discovered that they were capable of solving advanced problems otherwise impossible or less efficient for man to solve. Part of this capability is due to the computational machine's capability to iterate through and deal with large amounts of numbers and scenarios continuously and in a short amount of time. Soon, this aptitude gave way to the plausibility of artificial intelligence. In Alan Turing's landmark paper, "Computing Machinery and Intelligence," Turing developed his famous Turing Test, which verified the credibility of a computer being able to "think." Soon, artificial intelligence was implemented, first, in games. In 1951, Christopher Strachey of Manchester University created an AI for the game Checkers (Russel). However, the preliminary approach of accounting for complete domain knowledge using exhaustive conditional statements or mathematical models were either inefficient to program, yielded poor results, or were only applicable to specific games.

As a result, computer scientists took a more heuristic approach, creating systems that were more universally applicable. The general intuitively optimal and most exhaustive method of artificial intelligence for games with perfect information about all possibilities is Minimax. Minimax works by creating a complete game tree that holds all possible move sequences from the root or current state. Then, it explores each branch of possible moves to maximizes wins for the current player and minimizes losses by choosing the branch that leads to the most positive results. This is sufficient and perhaps optimal for simple games such as Tic-Tac-Toe and Checkers, but for even moderately complex games, these game trees become enormous and inefficient to iterate through computationally.

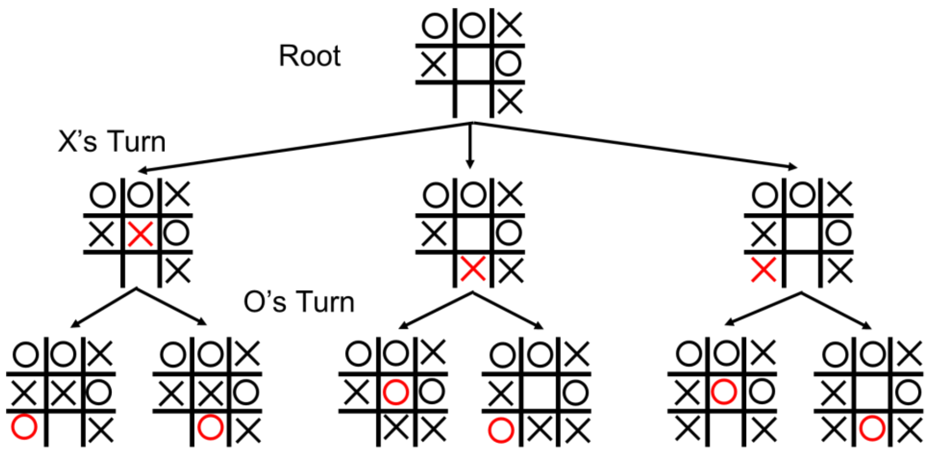


Figure 1: A portion of the complete game tree, illustrating the minimax process

The solution to this issue of enormous game trees is the Monte Carlo Method. The Monte Carlo method in computational mathematics is the use of random sampling to reach numerical data or extrapolated conclusions (Russel). In the case of Monte Carlo Tree Search (MCTS), it samples a predetermined number of branches or playouts of the game tree to reach conclusions rather than fully exhausting each and every branch like minimax. This requires much less memory and time to sample the tree incompletely and asymmetrically, making it very efficient. For example, in the game of Tic-Tac-Toe, an MCTS approach would iterate through n number of playouts, and choose the branch (move) that led to the most favorable outcomes (playouts that ended in wins). Selecting which branches to explore is dependent on the current selection policy being employed. This, along with the other ins and outs of the Monte Carlo Tree Search Algorithm will be discussed later [See Methods]. MCTS is especially useful because it entirely heuristic – the AI only needs the parameters of the game's movesets and crafts decisions from simulation based only on a few simple rules.

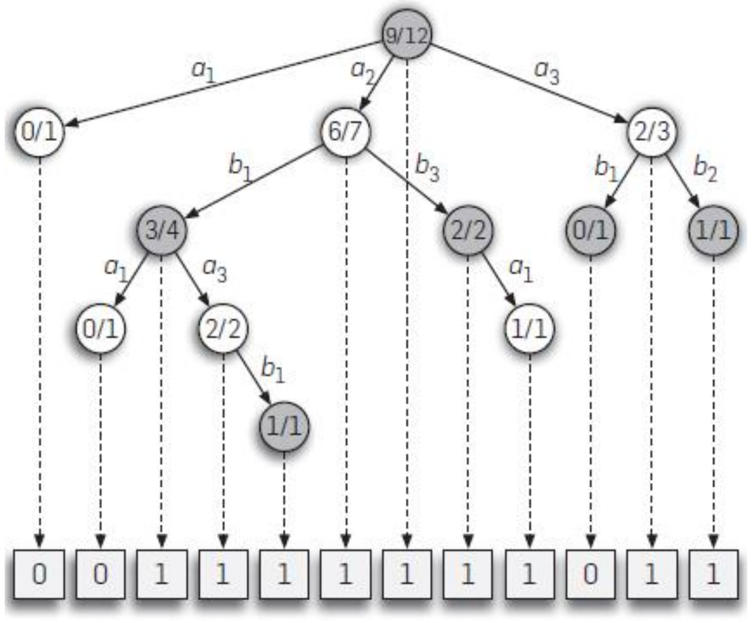


Figure 2: A barebones view of a MCTS tree

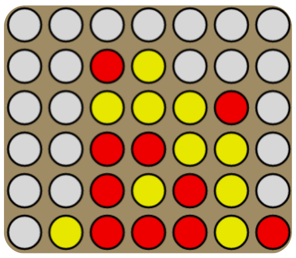


Figure 3: An example of a Connect Four game board

The game Connect Four is a classic two player logic game in which players must drop pieces in an upright 7x6 board and try to get four of their own color in a row, either diagonally or in a straight line. Given this game's 4,531,985,219,092 possible board permutations and currently undetermined number of terminal board permutations (currently estimated to be around 2 billion), applying the Minimax algorithm to it would be inefficient and infeasible. Thus, creating an artificial intelligence for Connect Four calls for a more efficient and heuristic, yet less complete, method of AI – Monte Carlo Tree Search. If successful, the versatile nature of Monte Carlo Tree Search could be expanded easily to a massive domain of games with tweaks to the code. To add, with mastery of this algorithm, there are numerous pratical real-world applications to extend its merits to (see further exploration in conclusion).

Goal/Objective

Program an efficient artificial intelligence using the Monte Carlo Tree Search Algorithm to make decisions on a turn-by-turn basis for the game Connect Four

Methods – Gameplay Framework

Materials. No external libraries or programs were used in this research. The only tools used were the Java Programming Language (Object-Oriented Approach) with JDK 1.8 along with the IDE Eclipse.

Program Design. The basic layout of the program was designed prior to engineering it. See **Figure 4** below for a topical visual overview. The basic flow of the program is as follows. First, the GameBoard is built, initially starting out empty at the beginning of the game. Then, the MCTS algorithm is run with the current GameBoard as the root node parameter. This algorithm is run for the pre-determined n number of iterations. Once the iterations are complete, the method will return the child branch with the highest value (there could be anywhere from 1-7 branches available on a 7x6 Connect Four Board). The GameBoard will execute the move of this child branch and check for a win or a tie. A win or tie would terminate this game, but otherwise, the player turn will change and the process cylces.

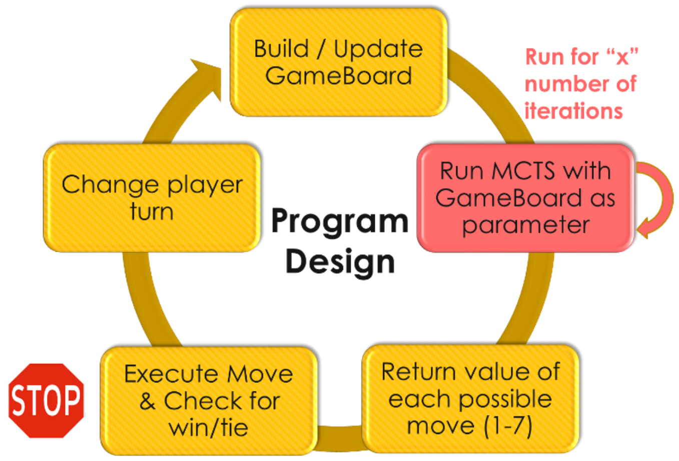
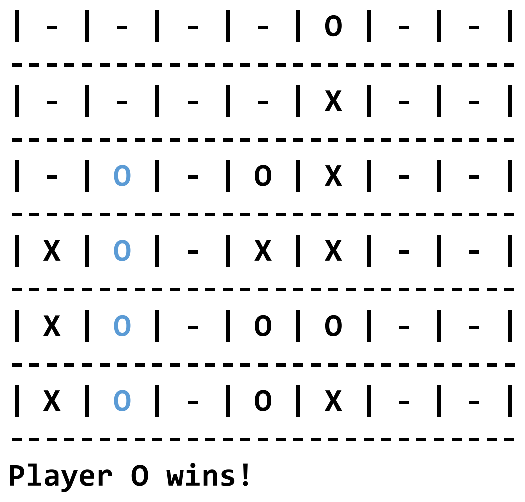

Build / Update GameBoard
Run for "x" number of iterations
Run MCTS with GameBoard as parameter
Change player turn
Program Design
Return value of each possible move (1-7)
Execute Move & Check for win/tie
STOP

Figure 4: Topical visual flow chart overview of the program design and process

GameBoard design. First, prior to implementing the AI, it was necessary to design a GameBoard class as the model and view of the program. This class would house necessary board configurations and implement the rules and nature of the game Connect 4. The board configurations were stored in 7x6 two-dimensional arrays (See **Figure 5**) and the pieces were denoted with either an 'X' character or an 'O.' A buildBoard method was also created in order to iterate through and print the array (See below or Line 97 of GameBoard.java).

```
char[][] gameBoard = new char [rows][columns]; //GameBoard 2D array
```

Next, a method was implemented in order to simulate making moves – assuring that each executed move was legal. In Connect Four, this means making sure that the column the piece is being 'dropped' in is empty, and making sure the piece falls to the bottommost row as the game implies that gravity will always bring the piece to the highest possible row index (lowest possible row physically). This piece input was capable of either artificial or human input. See Line 117 of GameBoard.java for code.



```
| - | - | - | - | O | - | - |
-----------------------------
| - | - | - | - | X | - | - |
-----------------------------
| - | O | - | O | X | - | - |
-----------------------------
| X | O | - | X | X | - | - |
-----------------------------
| X | O | - | O | O | - | - |
-----------------------------
| X | O | - | O | X | - | - |
-----------------------------
Player O wins!
```

Figure 5: Here is an example of a GameBoard output from the buildBoard method (below)

```
public static void buildBoard(char[][] boardState, char player)
      {

            System.out.println("");

            for(int i = 0; i < rows; i++){

                  for(int j = 0; j < columns; j++){
                        if (j == 0){System.out.print("| ");}
                        if (boardState[i][j] == '\0'){boardState[i][j] = '-
';} //fills empty spaces with '-'


                        System.out.print(boardState[i][j] + " | ");
                  }
            System.out.println("\n-----------------------------");
            }
      }
```

Next, it was necessary to create methods to check for wins each time a piece was played. This was done by iterating horizontally, vertically, and diagonally through the current board. If four pieces of the same type were found in a row, it was considered a win for that piece. Else, if each space on the board was filled and there were no such occasions of four pieces being in a row, it was considered a tie. See Line 187 (checkWin) and Line 80 (checkTie) of GameBoard.java for the methods.

<u>Methods – Monte Carlo Tree Search</u>

Overview. In order to decide which column to drop the piece in, the AI iterates through a set number of many playouts using the Monte Carlo Tree Search Algorithm as the sampling strategy. Then, using the results, some policy is used to decide which move from the current GameBoard (root node) leads to the most favorable results. The four main steps to Monte Carlo Tree Search are *Selection, Expansion, Simulation, and Backpropagation* (See **Figure 6**). This paper will go through each of their natures and implementations individually.

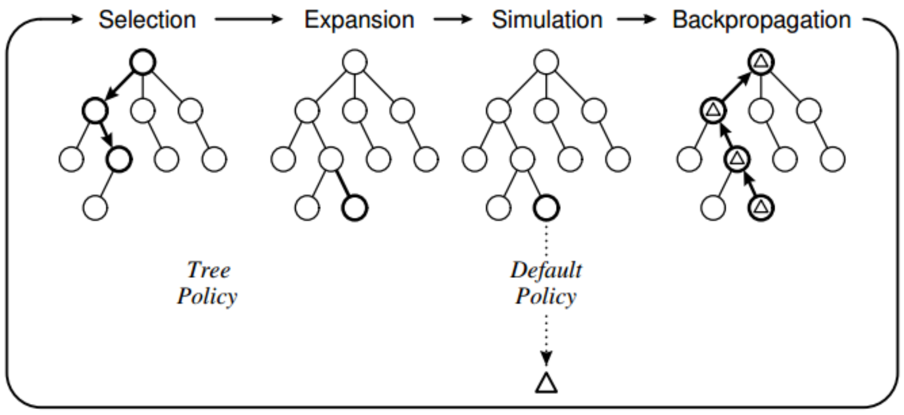
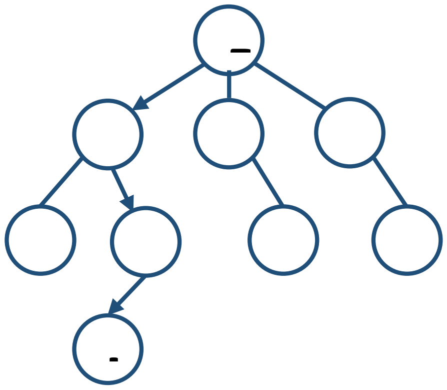


Figure 6: A barebones view of the Monte Carlo Tree Search Process. This is cycled through for each of many playouts.

In computational tree theory, each position or state on a tree (represented in these images as circles) are called nodes. The node that directly precedes another node is the parent, and the node that succeeds is the child, and the node that represents the current actual GameBoard state that MCTS iterates on is the root node. Each generation of children that is on the same row is known as a ply. Thus, a Node class was created in order to represent each instance of the Monte Carlo Tree. The instance fields of the Node class contain the current GameBoard state at that node position, the number of times that Node was visited (important for *selection*, updated during *backpropagation)*, the value of that node (important for *selection*, updated during *backpropagation*), and an array of the child nodes (if any) of the node in question.



Selection. In selection, the program begins from the root node provided by the GameBoard class and recursively selects child nodes using some policy until a node with no previously visited children is reached.

During selection, the main goal is to balance exploration and exploitation. This is so because when sampling the game branches of the complete game tree with MCTS, it is more beneficial to revisit branches that previously yielded promising results – exploitation. However, in order to assure that the search tree does not become exceedingly asymmetrical and all branches of the game tree are given an equal chance to be explored, the policy for selection must also account for searching a wider domain of branches – exploration.

A common, but very simplistic policy for selection is e-greedy. E-greedy always selects the child node with the highest value 1-n times, but chooses a random child node n times, where n is a small probability (e.g. 0.1). This way, this policy will always select the Node with the highest prior value (exploitation) except for a small percentage of the time by chance (exploration). See Line 310 of Node.java.

However, a more complex and effective policy often applied in MCTS implementation is the Bandit Policy (UCB), based on the *multi-armed bandit problem*. When combined with MCTS, the Bandit policy is known as Upper Confidence Trees (UCT) exemplified by this equation:

$$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

$$\bar{X}_j = average\ reward\ from\ arm\ j\ (value \div total\ visits)$$

$$C_p = tree\ exploration\ constant\ \left(generally\ \frac{1}{\sqrt{2}}\right)$$

$$n = \text{number of visits to the parent node}$$

$$n_j = \text{number of visits to child node (node in question)}$$

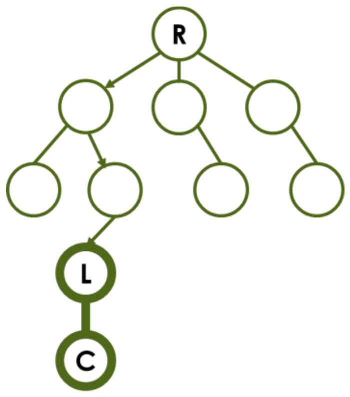
The properties of this equation naturally balance the essence of exploration and exploitation. Firstly, $\bar{X}_j$ results from dividing the value of a node by its number of visits; thus, if a node has been visited excessively, its average reward will diminish. A similar process takes place under the radical when n is divided by nj. Additionally, the tree exploration constant assures that the level of exploration is consistent and existent for all iterations. When an implementation using UCT was played against e-greedy, the UCT implementation won for all turnouts with playout iterations >= 500.

The selection method was implemented in Line 280 of the Node.java file (see **Figure 7**). It uses an enhanced for loop to iterate through all children and apply the UCT formula to each. In the end, the child which resulted in the highest value from the UCT value is returned. Some nuances that had to be accounted for include that unvisited nodes resulted in divide by zero errors. Essentially, these nodes were unexplored and approached a UCT value of ∞, but needed to be accounted for in the program. The method adds the number of visits to a node with a small *buffer* value in order to simulate zero as a very small number. In the case that a node with no previously visited children is reached (advent of *expansion*) the selection method also adds a very small random number in order to choose the child randomly.

```
private Node select() //Upper Confidence Bound (Bandit Policy) implemented to
select most optimal node based on exploration-exploitation balance
	{
		Node selected = null;
```

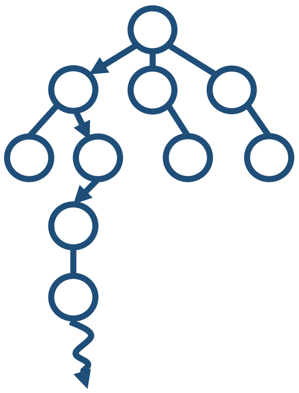
```
            double bestUCB = 0;
            for (Node childNode : children)
            {
                  if (childNode == null){continue;} //for the special case of
                  firstIteration children
                  // UCB = estimated value(average reward) + C *
sqrt(ln(number of visits to parent) / (number of times (child) node has been
visited))
                  // Where C is some constant -- sqrt(2) in this case
                  // buffer prevents divide by 0 errors in the case of
unvisited nodes
                  double ucbValue = childNode.nodeWins /
(childNode.nodeVisits + buffer) //estimated value
                        + (2/Math.sqrt(2)) * Math.sqrt(2 *
Math.log(nodeVisits+1) / (childNode.nodeVisits + buffer)) +
                    randomNumb.nextDouble() * buffer; //generates a very
small random number to break ties between similar/unvisited nodes
                  if (ucbValue > bestUCB)
                  {
                        selected = childNode;
                        bestUCB = ucbValue;
                  }

            }

            return selected;
      }
```

Figure 7: Selection method from Line 280 of Node.java



Expansion. Expansion picks up from the final node of selection with previously unvisited children, and creates children for that node and chooses one. This choice will always be random because the children do not have stats to distinguish them as they are unvisited and newly created.

See Line 227 of Node.java for the expansion mutator method. It takes the parameters of a board state and the list of the current node's children (which passes as empty). It uses two nested for loops to iterate through each possible move from the given node. Each column of the board has at most one possible move as it always falls to the lowest row. Once that lowest row is found (if the column isn't full) it creates a clone of the current parent nodes board and adds the new piece. It then creates a new child node in the parent node's list. Each node has between 0 and 7 children due to the GameBoard size, where 0 is the result of a full board. In any step of the MCTS algorithm, it ends if it reaches a terminal node.

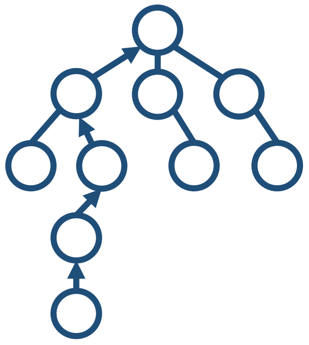


Simulation. Starting from the child node chosen from expansion, simulation plays out (or *simulates*) a completely random game until some terminal result is reached – either one player wins or the board is completely filled and it is a tie.

The simulation method was implemented simply from a while loop switching back and forth between expansion and selection – creating children for the current node and choosing a random child (recall the select method does this when all the children are unvisited). Additionally, an extra opponent-expansion method must be implemented in order to randomly simulate opponent moves in between the current player's moves. The opponent's moves are all selected completely randomly with no foresight into the opponent's knowledge.

```
//Simulation/Expansion: simulate (random) game until there is a winner or
there is a tie.
            while (!GameBoard.checkWin(currentNode.boardState, 'O')
                         && !GameBoard.checkWin(currentNode.boardState, 'X')
                         && !GameBoard.checkTie(currentNode.boardState))
            {
                  currentNode.expansion(currentNode.boardState,
currentNode.children, firstIteration); //Expands a given node to create child
nodes for the current players turn
                  firstIteration = false; //after any node has expanded, it
is impossible for it to be the root node
                  currentNode.oppExpand(currentNode.boardState,
currentNode.children); //Simulates all possible random opponent moves as
children of players possible moves
                  currentNode = currentNode.select(); //selects a player
child node to continue simulation (randomly since none of these nodes have
been visited yet as they are newly made)
                  if (!currentNode.oppMove) {visitedNodes.add(currentNode);}
                  if (!GameBoard.checkWin(currentNode.boardState, 'O')
                              && !GameBoard.checkWin(currentNode.boardState,
'X')
                              && !GameBoard.checkTie(currentNode.boardState))
                  {
                        currentNode = currentNode.select(); //selects
opponent child of player children randomly
                  }
            }
```

Figure 8: Simulation loop (continuously checking for playout termination; firstIteration Boolean preserves indexes for root nodes during move selection process.



Backpropagation. Finally, after the playout has reached a terminal state, backpropagation is the step that backtracks through all the visited nodes and updates their stats. In order to do this, a visitedNodes list is kept throughout the duration of the AI, and this phase uses an enhanced for loop to iterate through each one.

During backpropagation, two specific things need to updated in order to update the UCT results for the next playout iteration. One is that each node receives one additional visit count. Next, depending on the terminal result in simulation, each node gets a value. There are an infinite ways to assign values to the nodes. The general trend is to assign maximum values to playouts that terminate with wins, minimum values to losses, and medium values to ties. Because node values are used as ratios to the number of visits in the UCT formula, the exact number attributed to the value does not matter as much as the ratio of all the values (i.e. 0, .5, 1 to losses/ties/wins respectively is *similar* to 0, 1, 2).

There are multiple options for value assignment (losses/ties/wins):

Balanced Play (All results are distributed evenly) → 0/0.5/1

Offensive Play (All non-win entities are treated as less) → 0/0/1

Defensive Play (Focuses on not losing rather than winning) → 0/1/1

Highly Defensive Play (Would rather tie than lose or win) → 0/2/1 (or any extreme value for ties)

Each of these options have many variants, but these are ones in question for testing purposes.

Implementing backpropagation is quite simple – simply create a conditional statement for each possible turnout (loss, win, tie), and updating respectively (See Line 102 of Node.java).

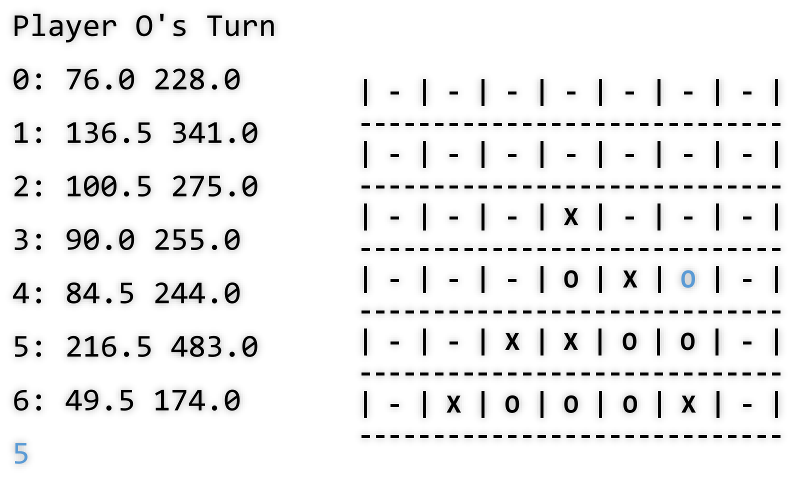
Making the Move. Once all n number of iterations for the MCTS algorithm are complete, a move must be made. There are three conventional policies for choosing which move to make based on the collected data:

Max child: Choose the root child with the most wins

Robust child: Choose the root child with the most visits

Max-robust child: Choose the child with the greatest ratio of wins to visits

For the game Connect Four in combination with the UCT formula, each of these three policies will virtually always select the same root child, thus the policy choice is negligible. After the MCTS iterations are complete, a for loop simply iterates through all the possible moves (root children) and chooses the one with the greatest average reward (See Line 279 of GameBoard.java). Then the players turn changes and the process repeats with the new root node. (See **Figure 9** for sample output)



```
Player O's Turn
0: 76.0 228.0       | - | - | - | - | - | - | - |
                    -----------------------------
1: 136.5 341.0      | - | - | - | - | - | - | - |
                    -----------------------------
2: 100.5 275.0      | - | - | - | X | - | - | - |
                    -----------------------------
3: 90.0 255.0       | - | - | - | O | X | O | - |
                    -----------------------------
4: 84.5 244.0       | - | - | X | X | O | O | - |
                    -----------------------------
5: 216.5 483.0      | - | X | O | O | O | X | - |
                    -----------------------------
6: 49.5 174.0
5
```

Figure 9: Sample of program output. Gives value of each column (left) and number of times it was visited (right). It then chooses the column of highest value (index 5) and drops piece there and builds GameBoard to display.

<u>Methods – Testing</u>

An effective alternative way to test this AI may be with human subjects. However, with the lack of resources and time to develop sufficient controls to reach adequate results, a more qualitative and computational approach to data collection was taken.

MCTS vs. MCTS Playouts. The main method of testing this program was by playing two MCTS AIs against one another with varying levels of playout iterations. This was done for all combinations of MCTS with 500, 1000, and 5000 playout iterations. As a control, each of these matchups used the *Balanced Play* backpropagation policy. Data was taken for 1000 game results between each of these matchups as either a win, a tie, or a loss. This data collection was automated by continuously playing playouts in a while loop for 1000 games and writing results to an Excel graph using a modified code.

Initial Move Speed. Another test was to use *System.nanoTime()* in order to time just the initial move of the AI (starting from empty root node gameboard) in seconds. Only the initial move was tested because as the GameBoard was gradually filled, the MCTS playouts grew exponentially faster as simulating from an empty GameBoard takes much longer as it would take longer to reach a terminal state. Thus, the initial move of an AI in a game supplies a better benchmark speed.

Backpropagation Policy. In order to test the value update (play-style) policies for *backpropagation*, MCTS AI (control of 1000 playouts) were tested against one another with wins, losses, and ties as the categorical criteria. The variable would be the policy used.

MCTS vs e-greedy Policy. MCTS at 500, 1000, and 5000 playouts were played against e-greedy at 0.1 and 0.2 n-values at 5000 iterations. Results were recorded as wins, losses, or ties. (See *Selection* Section for details on e-greedy).

MCTS vs Minimax. This is a theoretical yet implausible test as the sheer magnitude of the Connect Four complete game tree is too large for the household computer to iterate through. Thus, this test could not be done, however it is predicted that Minimax would perform equally or better than MCTS, but it would take *much* longer.

Results

Table 1: MCTS AI vs AI Turnouts (1000 Trials for Each Matchup)

	5000 vs 5000	5000 vs 1000	5000 vs 500	1000 vs 1000	1000 vs 500	500 vs 500
Player 1 Wins	355	573	798	353	567	411
Ties	220	51	54	280	124	222
Player 2 Wins	425	376	148	367	309	367
Totals	1000	1000	1000	1000	1000	1000

Figure 10: MCTS AI vs AI Turnouts (1000 Trials for Each Matchup) [Table 1]

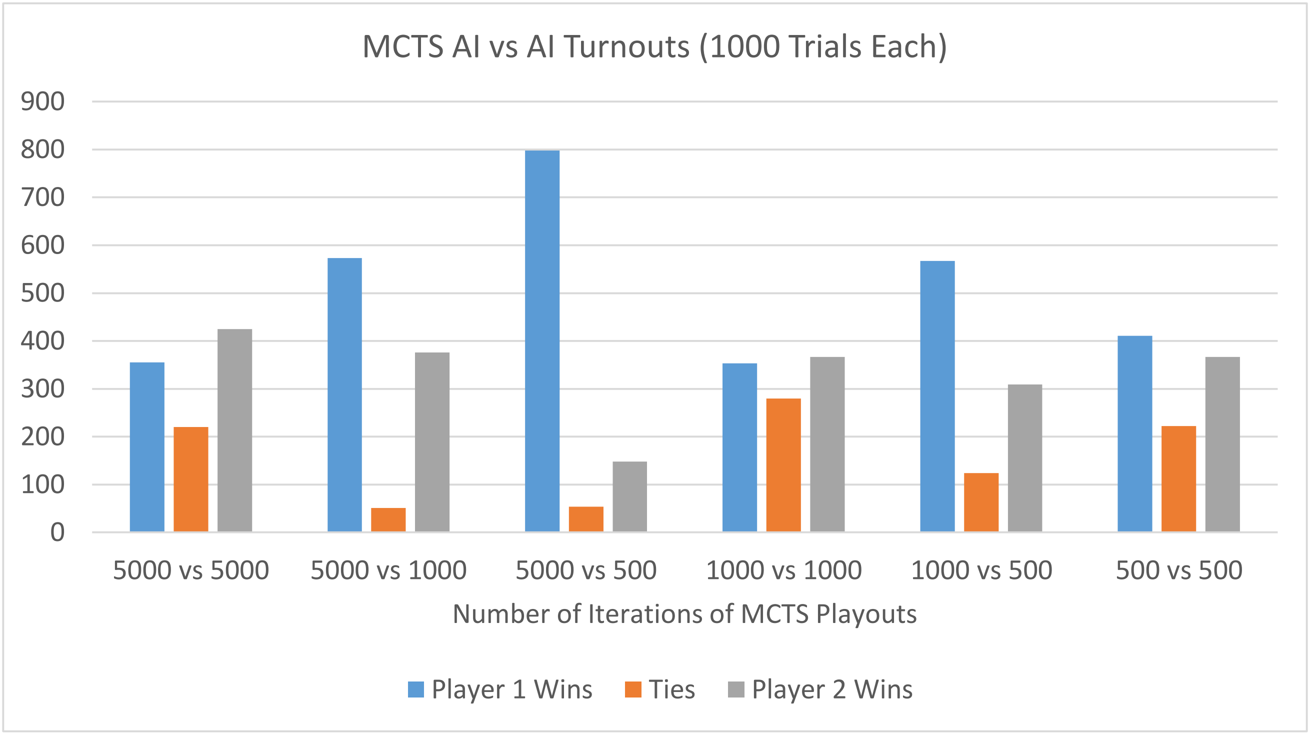


Table 2: Time Elapsed for Initial Move

Playout Iterations	Time Elapsed for Initial Move (Average of 1000)
500	0.306
1000	1.014
2000	1.974
3000	2.591
4000	4.100
5000	5.240

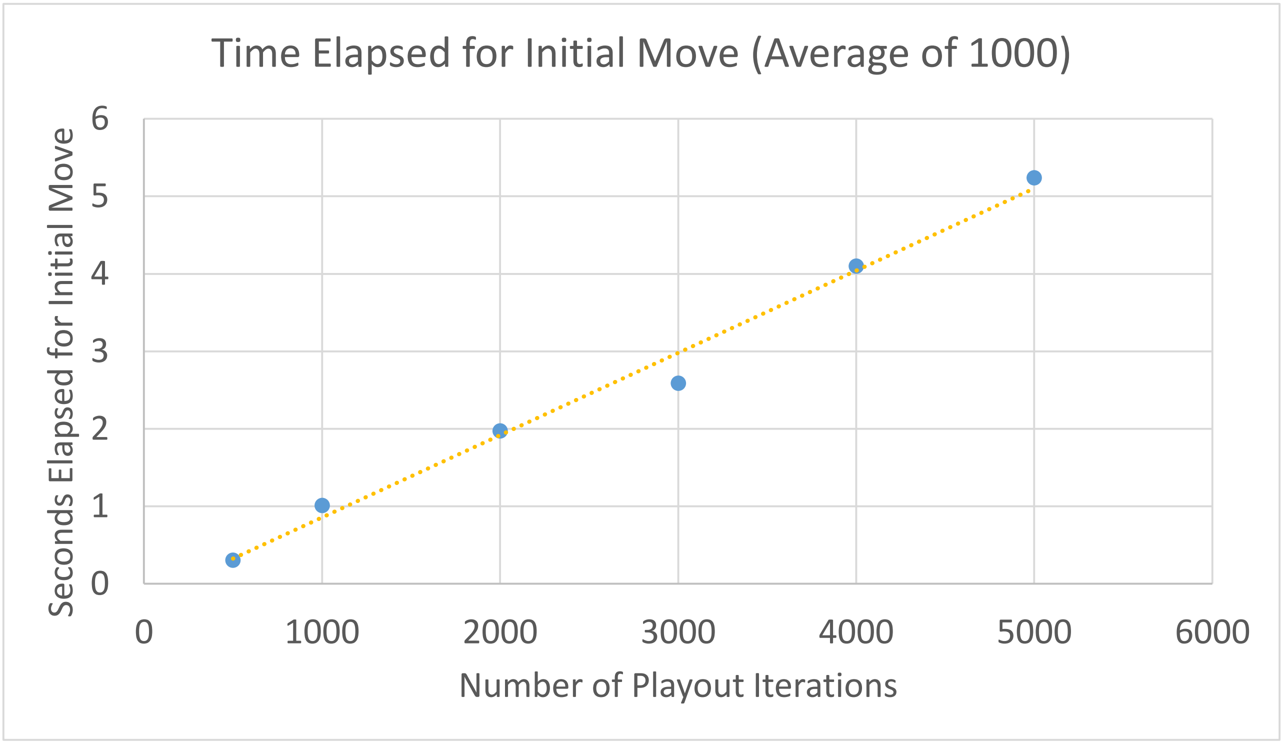


Figure 11: Time Elapsed for Initial Move (with LSR Line) [Table 2]

<u>Estimating Time Elapsed for Minimax</u>

LSR Line approximation → y seconds = x playouts * .001

y = 2000000 seconds

y = ~23.148 days

Table 3: E-greedy (5000 iterations) vs MCTS Playouts (Total of 1000 trials for each matchup)

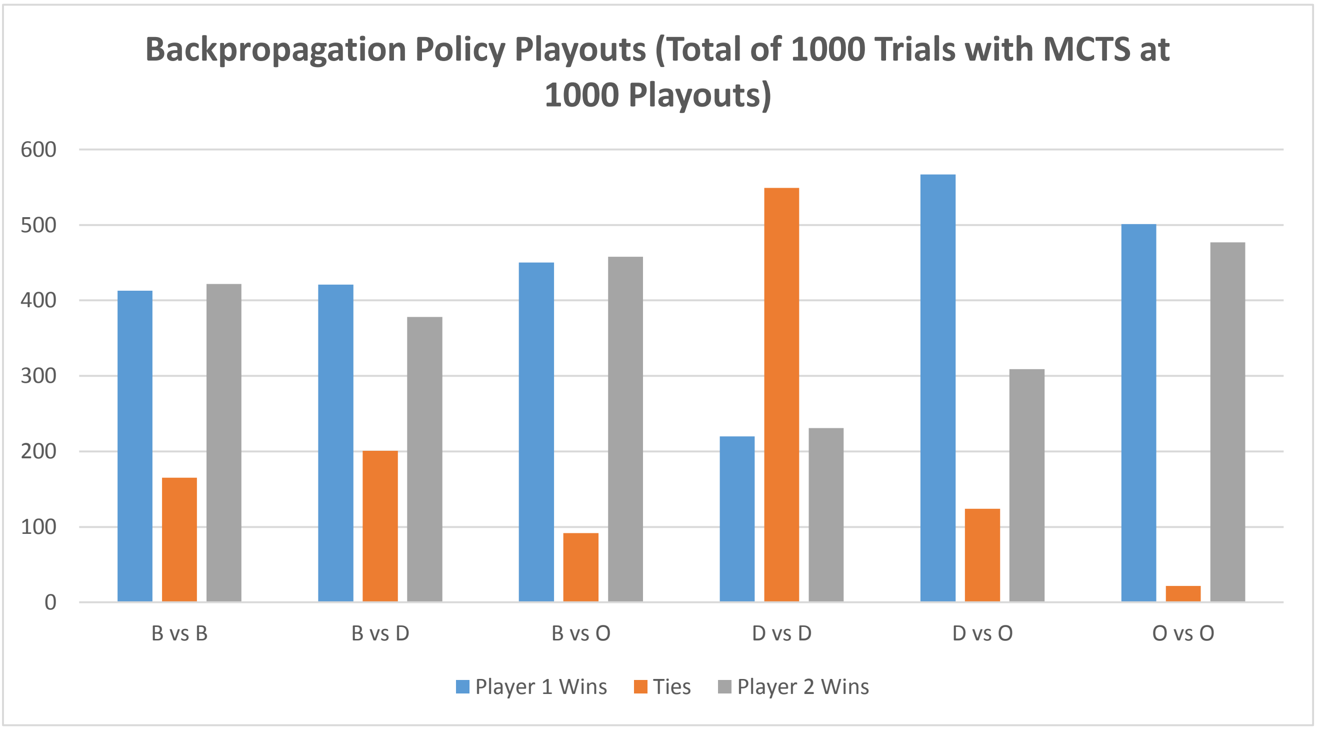
	1000 vs n = 0.1	1000 vs n = 0.2	500 vs n = 0.1	500 vs n = 0.2	5000 vs n = 0.1	5000 vs n = 0.2
MCTS Wins	1000	997	978	953	1000	1000
Ties	0	3	22	45	0	0
E-Greedy Wins	0	0	0	2	0	0
Totals	1000	1000	1000	1000	1000	1000

Table 4: Backpropagation Policy Playouts (Total of 1000 Trials with MCTS at 1000 Playouts)

B = Balanced (0/.5/1)
D = Defensive (0/1/1)
O = Offensive (0/0/1)

	B vs B	B vs D	B vs O	D vs D	D vs O	O vs O
Player 1 Wins	413	421	450	220	567	501
Ties	165	201	92	549	124	22
Player 2 Wins	422	378	458	231	309	477
Totals	1000	1000	1000	1000	1000	1000

Figure 12: Backpropagation Policy Playouts (Total of 1000 Trials with MCTS at 1000 Playouts) [Table 4]



<u>Discussion – Analysis</u>

First, the MCTS AI vs AI playouts yielded some interesting implications (**Table 1/Figure 10**). As expected, those MCTS AIs with a higher number of playout iterations performed much better than those with lesser playout iterations. This is evident because the AI with 5000 iterations received 197 more wins than the 1000 iteration AI and 650 more wins than the 500 iteration AI when played against one another (both matchups had a negligible number of ties). Also, when AIs of the same playout iterations were played, they had equal success. Thus, it is apparent that a greater number of playout iterations means a more successful MCTS AI.

However, it seems that the relationship between AI success and playout iterations is not linear. This is so because although the 5000 playout AI had 4000 more iterations than the 1000 playout AI, it did not do substantially better (enough to propose a linear relationship). Therefore, it may be more appropriate to model this relationship with a logarithmic or inverse tangent model as the initial increase in playout iterations has a drastic effect on MCTS success (as evident between 1000 vs 500), but this increase in success slowly levels off logarithmically or asymptotically as it approaches a large amount of playout iterations.

Furthermore, as the number of playout iterations increases, so does the inefficiency and time required by the AI to make a move (**Table 2/Figure 11**). This is illustrated by the amount of time it takes for an initial move to be made given the number of playout iterations of MCTS. As it is apparent in **Figure 10**, the relationship between playout iterations and the time elapsed for the first move is linear. Moreover, when this information is combined with the previously derived information that higher playouts numbers do not do substantially better than lower ones, it may be a likely conclusion that for the game Connect Four (and perhaps others) there is an optimum number of playout iterations of MCTS. This is so because it may be inefficient to always use the maximum number of playout iterations since the time elapsed increase linearly/indefinitely, yet its success only increases by a slight margin.

The E-greedy data shows that MCTS is indubitably superior to the simplistic e-greedy policy for the game Connect Four (and likely other games as other evidence shows). Even the most limited number of MCTS iterations (500) had no trouble in completely thwarting 5000 iterations of e-greedy at both values of n. This shows that the MCTS UCT Bandit-based selection policy is far superior in its efforts to balance exploration and exploitation. Additionally, although the minimax implementation was far too inefficient to test, extrapolating the results from **Figure 11** to account for the 2 billion playouts (where y seconds $\approx$ x/1000 and x = 2 billion playouts) of minimax estimates the initial move would take approximately three weeks (~23 days), much longer than MCTS (See page 13).

Lastly, the backpropagation policy results illustrate various trends as a result of changes in policy. For one, the balanced policy seemed to always end in matching the number of wins as the opposing policy, proving it to be consistently robust. To add, when two AIs with the same policy were played against one another, they always resulted in approximately the same number of wins. One unique feature is that when the two defensive strategies were played against one another, it almost always resulted in a tie. This may be useful knowledge given that Connect Four is a game in which when two players play ‘perfectly’ it is possible to always tie (Kocsis). Thus, it may be that the defensive strategy is the most optimal for ‘perfect’ play. To add, when the defensive was matched up against offensive it was the only strategy to come out with substantially more wins.

Discussion – Future Implications

The strengths in this objective include that the data consistently replicated itself and followed predictable patterns that correlated with theoretical/prior data for the most part. In other words, although a very large sample of 1000 trials was taken for each method of data collection, similar results would be replicated with much smaller sample sizes as well.

The shortcomings of this program are usually a result of natural sampling error in the MCTS playouts. For example, sometimes the AI will fail to block a clear opponent's impending win (especially in less conservative offensive backpropagation policies) that may be very apparent to humans. This may be because, due to random probability, the playouts of the current MCTS AI failed to find the branch that had this as an outcome. Additionally, it may be that all the other branches the AI tested found losses later down the road as well so it doesn't matter. In this sense, MCTS is blind to playout depth, whereas the human mind is more prone to see dangers in closer proximity. One way to account for this issue may be to add domain knowledge to the AI. Although this may be corruptive of pure MCTS's heuristic use of perfect information, it may be interesting to see what results would ensue if the AI was given information about instant or established Connect Four strategies. Another goal would be to optimize the code to get the AI speeds to be even quicker and allow for a greater number of iterations.

Continuous goals, beyond cleaning up the efficiency and readability of the code include further optimizing the configurable portions of the AI. A current/future endeavor is to try to find the aforementioned optimal number of AI playout iterations in order to balance time efficiency and AI success (if one exists). To add, another variable to optimize is the backpropagation policies. Currently, these policies are only being tested categorically (based on relative relationships between the loss/tie/win values) as balance, defensive, or offensive. A more conclusive method may be to study these policies mathematically or use some iterative method to try many different combinations to gather results for a greater range of quantitative policies.

Conclusion

Overall, the initial goal to program an efficient artificial intelligence using the Monte Carlo Tree Search Algorithm to make decisions on a turn-by-turn basis for the game Connect Four was achieved. The end result was a working implementation of the classic game Connect Four in which the MCTS AI showed considerable levels of success. It was found that as the number of MCTS playout iterations was increased, a greater success (number of wins) of the AI was found. Furthermore, it was also found that that as the number of iterations was increased, the efficiency of the AI decreased. However, this implementation of AI was determined to much better than the impossible minimax and the simplistic e-greedy policy. Also, MCTS's versatile backpropagation policies make it capable of multiple strategies. Although there is much future improvement to be made, this success with Connect Four illustrates some of the tendencies of MCTS to apply to a greater range of games in the future as well as other applications beyond games.

There are numerous practical applications that this research in MCTS could extend to. MCTS simply requires a situation in which perfect information is available about the domain of possible ‘moves’ that can be made. From this point, MCTS uses its heuristic methodology to reach conclusions for any situation. Thus, MCTS has many practical applications if its basic process is tweaked to the parameters of the limits of a given situation. For example, given possible outcomes in a stock market, a MCTS algorithm could sample and select stock related decisions. Furthermore, in more complex situations, MCTS could also be applied to robot decision making. Moreover, a combination of the field of computer science and biology is found in bioinformatics. One of the current most pressing issues in bioinformatics is analyzing copious amounts of DNA information as well as discovering a seemingly unlimited combination of protein structures. If an MCTS artificial intelligence was given domain in the form of the tendencies of amino and nucleic acid bonds and formations, a program could very likely sample and move forward in building polymerizations that could prove useful to humans.

References

Browne, Cameron B. "A Survey of Monte Carlo Tree Search Methods." *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012). Print.

Chaslot, Guillaume. *Monte-Carlo Tree Search: A New Framework for Game AI*. Masstricht: Universiteit Maastricht. Print.

Kocsis, Levente. *Bandit Based Monte-Carlo Planning*. Budapest: Computer and Automation Research Institute of the Hungarian Academy of Sciences. Print.

Remi Munos. From Bandits to Monte-Carlo Tree Search: The Optimistic Principle Applied to Optimization and Planning. 2014.

Russell, Stuart J., Peter Norvig, and Ernest Davis. *Artificial Intelligence: A Modern Approach*. 3rd ed. Upper Saddle River: Prentice Hall, 2010. Print.