Monte Carlo Tree Search and Tic-Tac-Toe

Introduction: What is the Monte Carlo Tree Search Algorithm?

Monte Carlo tree search (MCTS) is a decision-making algorithm most often used in artificial intelligence software that play board games. One notable usage is in Google's AlphaGo AI, which famously beat the number one Go player in the world, Ke Jie, in 2017. Other uses of the MCTS include software for playing checkers and chess. Even Tesla's Autopilot Software incorporated this algorithm.

It is particularly effective for "perfect-information" games, where all the information regarding the progression of the game is available to both players. Go, chess, and checkers are all examples of perfect-information games. It does not do so well with games with imperfect information, such as poker, where the hands are hidden and players can bluff (thus hiding their true intentions).

Background: How does the MCTS work?

MCTS decisions are heuristic, meaning we are not guaranteed precisely the correct or the best decision, but we can get an approximation that can often be good enough. As a general overview, this algorithm will generate a game tree, with a node for each state of the game. As the algorithm plays the game, it will track the sequence of game states (or statuses of the board after a move is made). Upon reaching an end result, it will propagate back up the sequence and update the states with the win rate. Over time, the algorithm will generate a tree with the success rates for making each move to the next state.

Essentially, we are making educated guesses on which move to make based on the state of the game and the outcomes from previous play-throughs. The more experienced a game tree is, the better the algorithm performs.

When we are training the algorithm, we make predictions using the UCB formula. The algorithm will predict the node with the highest UCB value as the most favorable and explore that one. The formula generates a value that favors nodes with the highest success rate, but also leaves room for exploring other nodes.

The formula:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

In which

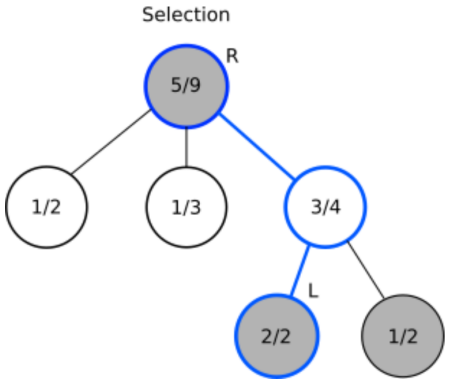
- w_i = number of wins after the i-th move
- n_i = number of simulations after the i-th move
- c = exploration parameter (theoretically equal to $\sqrt{2}$)
- t = total number of simulations for the parent node

Each node has a UCB value. Say the algorithm is at node N. First, we will redefine the terms in a way that is easier to understand:
w = the number of wins that came from after making a move to node N
n = the number of times node N has been visited
c = the exploration parameter, which is a constant that can be adjusted based on how much the implementer wants the algorithm to explore other nodes
t = the number of visits to node N's parent, or the number of times the algorithm had to choose between node N and its siblings

In the formula, the left term is the exploitation term (which indicates how successful a move is), and the right term is the exploration term (which indicates how often we make a certain move). We want a balance of choosing what works and exploring new options. A node that has a high success rate would have a high exploitation term value. At the same time, a node that has been explored very little would have a very high exploration term value (if the game state is at node P and infrequently chooses its child node N, then the t term would be very high and the n value would be very low).

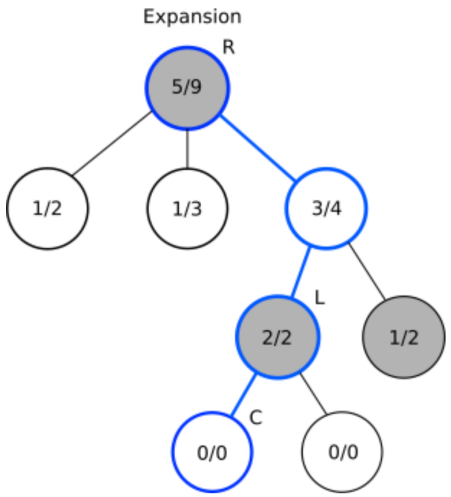
After the game tree has gained enough experience, during the actual testing, it will choose the move to the node that has been visited the most, since the algorithm chooses the most successful moves the most.

There are four stages in a MCTS step:
1. Selection:



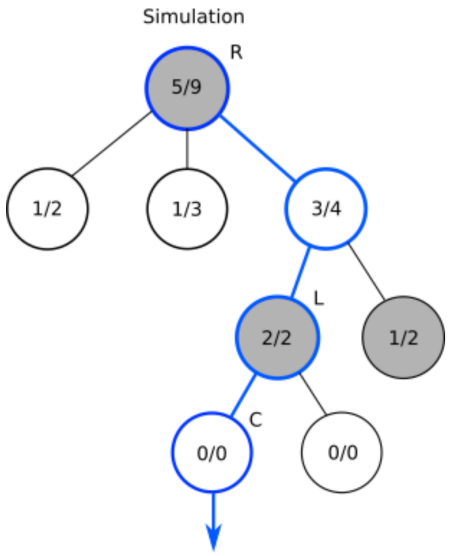
We start at the root node, R, and we select successive child nodes with the highest UCB value until we reach a leaf node L. Here, the exploration term is left out. There is likely a different method of picking between the exploitation of previous runs and exploration. Whatever the case, the UCB value is dominated by the success rate anyway.

2. Expansion



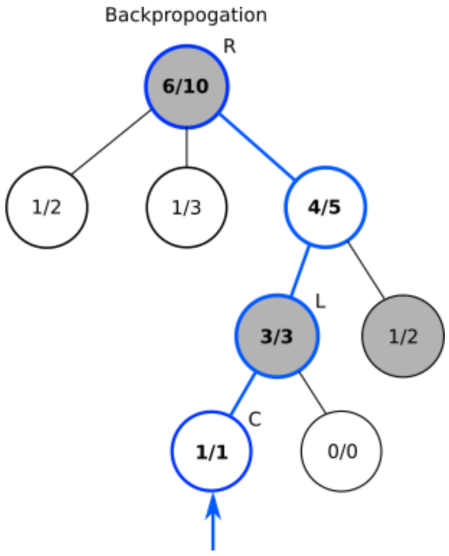
Unless the L node results in an end game state, we then go into the expansion phase and create all possible outcomes from L's game state and randomly select one of L's child nodes, C.

3. Simulation



In the simulation phase, we generate a random playout from node C to an end game state.

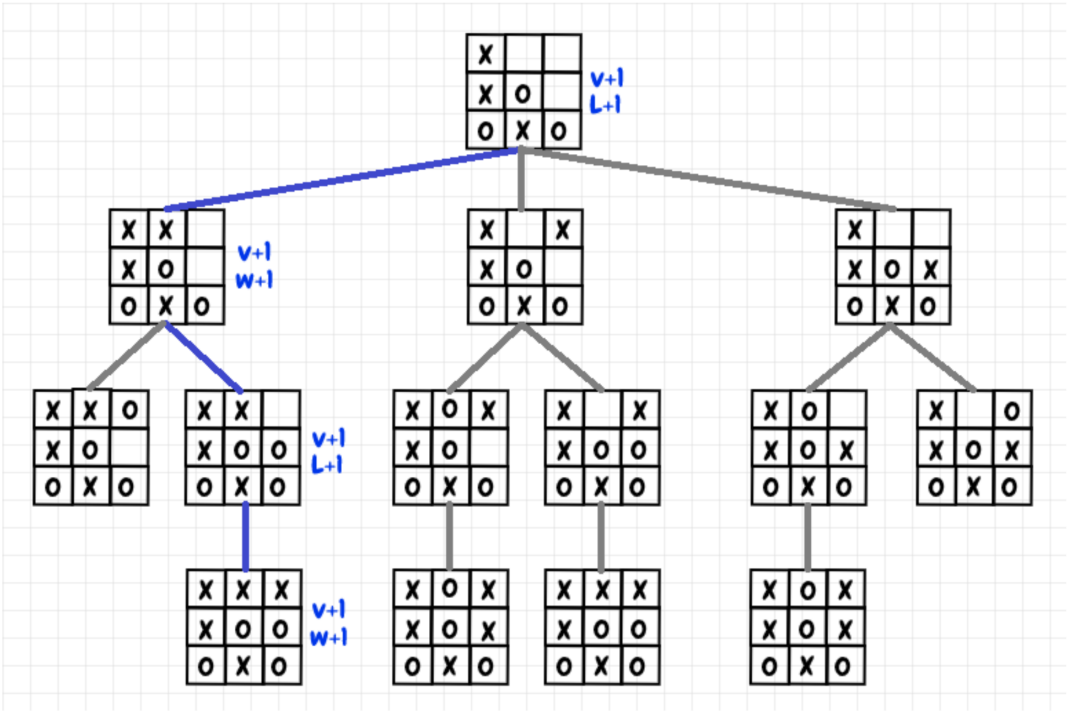
4. Backpropagation



During back-propagation, we use the resulting game outcome to update the information within the nodes from C to R.

The algorithm will cycle through all 4 phases until the allotted number of iterations has passed to add experience to the tree. Afterwards, it will choose the child node of the root R with the highest amount of visits (the more visits mean more selections, which only occurs if it has the highest likelihood of winning). This then repeats for each turn of the game, with chosen child node becoming the new root R.

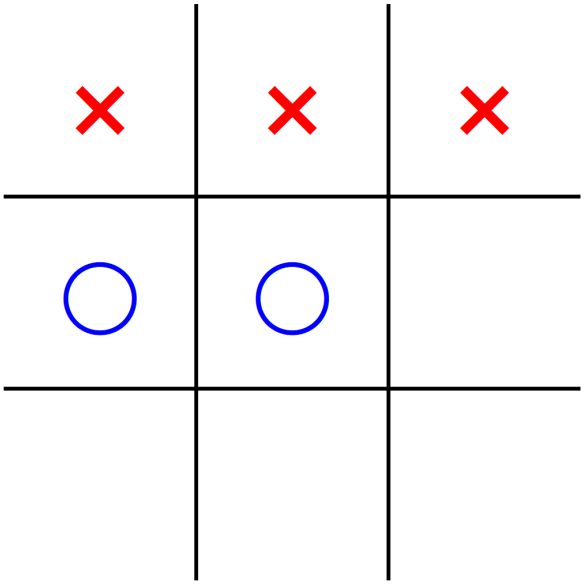
Implementing MCTS for Tic-Tac-Toe



Above is a version of the MCTS for tic-tac-toe, which is slightly different from the version in this notebook. However, the fundamental idea is still present: play through an entire game and backtrack the moves taken with the end result. In the image, the visit count is incremented for each game state. Then based on whose turn it was to REACH a given node, either the win or loss count is incremented. For example, it was X's move that placed the game to the left-most node of the second row. Because X won, update the win count for that node.

Before we get into implementing the MCTS algorithm, there are a few clarifications to be made:

A board is represented as a flattened list. So, {"x", "x", "x", "o", "o", "b", "b", "b", "b"} represents the following board state:



A node in the the search-tree is represented with a replacement rule list.
{parent -> n, board -> b, children -> {...}, visitCount -> v, winCount -> w, player -> p}.

Player p is the player to make the next move on the current board b. Player 1 plays X, Player 2 plays O.

Selection Phase

```
In[•]:= findUCB[ t_, index_] := Module[{w , v },
          (* SPECIFICATION: Returns the UCB value of the node with the input index. *)

          w = winCount /. n[index];
          v = visitCount /. n[index];
          (* If the node has never been visited,
          then the UCB value is set to the highest integer value
           (meaning the w/v term has v = 0, and we want to check out this node that has
              never been explored). Otherwise, the UCB is computed using the formula. *)
          If[v == 0, $MaxNumber, N[w / v + Sqrt[2] * Sqrt[Log[t] / v], 3]]
         ];

        findBestUCBChild[index_] :=
          Module[{numVisits, childrenList, bestNodeIndex, max, currNodeIndex, currUCB, i},
           (* SPECIFICATION: Returns the child of the node
             with the input index with the best UCB value. *)

           numVisits = visitCount /. n[index];
           (* This is the "t" value in the UCB formula. *)
           childrenList = children /. n[index];

           (* We go through the node's children,
           and we save the index of the one with the highest UCB value. *)
           bestNodeIndex = childrenList[[1]];
           max = findUCB[numVisits, bestNodeIndex];
           For[i = 2, i ≤ Length[childrenList], i++,
            {
             currNodeIndex = childrenList[[i]];
             currUCB = findUCB[numVisits, currNodeIndex];
             If[currUCB > max, {max = currUCB; bestNodeIndex = currNodeIndex;}, Null];
            }
           ];

           bestNodeIndex
          ];

        selectPromisingNode[index_] := Module[{childrenList},
           (* SPECIFICATION: Select the index of the most
             promising node to explore based on the highest UCB value. *)

           (* Select one of the input node's children to explore. If there are no children,
           the input node will be explored. *)
           childrenList = children /. n[index];
           If[Length[childrenList] > 0, findBestUCBChild[index], index]
          ];
```

Expand Phase

```
In[•]:= generateNextBoards[board_, player_] :=
          Module[{move, nextBoards, blankSpaces, newBoard, i},
```

```
    (* SPECIFICATION: Generates a list of the next
      possible boards one turn after the input board. *)

    If[player == 1, move = "x", move = "o"]; (* Player 1 plays "x", Player 2 plays "o". *)
    nextBoards = {};
    blankSpaces = {};

    (* Saves a list of the indices of blank spaces in the input board *)
    For[i = 1, i ≤ 9, i++, If[board[[i]] == "b", blankSpaces = Append[blankSpaces, i], Null]];

    (* Use the list of blank space indices and to add a move
     to the input board. Each possible next move becomes a new board. *)
    For[i = 1, i ≤ Length[blankSpaces], i++,
     {
      newBoard = board;
      newBoard[[blankSpaces[[i]]]] = move;
      nextBoards = Append[nextBoards, newBoard];
     }];

    nextBoards
   ];

expand[index_] := Module[{currBoard, currPlayer, childrenList,
    nextBoards, newBoard, newIndex, oldParentIndex, currIndex, i},
   (* SPECIFICATION: Creates the children for the node of the input index. *)

   currBoard = board /. n[index];
   currPlayer = player /. n[index];
   childrenList = {}; (* We only expand leaf nodes, or nodes without any children. *)
   nextBoards = generateNextBoards[currBoard, currPlayer];

   (* For each possible board with a new move,
   create a new node and add its index to the children list for the node with
    the input index. If the node from the input index is an end game state,
   there will be nothing in the nextBoards list. *)
   For[i = 1, i ≤ Length[nextBoards], i++,
    {
     newBoard = nextBoards[[i]];
     newIndex = StringJoin[newBoard];

     (* Define a new node for the next move and set the parameters. The
      player field is set to the opponent of the current player. The new
      board is updated with the most recent move of the current player,
     so the opponent has the next move. Add the new node to the children list. *)
     n[newIndex] = {parent → index, board → newBoard, children → {},
       visitCount → 0, winCount → 0, player → 3 - currPlayer};
     childrenList = AppendTo[childrenList, newIndex];
```

```
    }
   ];

   (* Replace the children list
    for the node of the input index with the updated one. *)
   n[index] = n[index] /. (children → {}) → (children → childrenList);
  ];
```

Simulation Phase

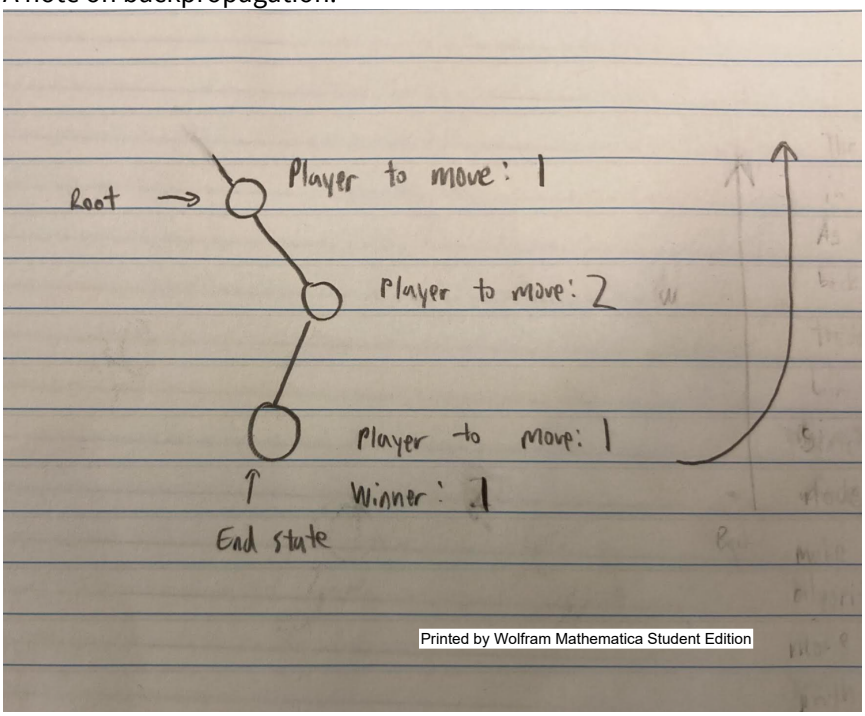
In[ ]:=
```
checkRowWin[board_] := Module[{},
   (* SPECIFICATION: Returns 0 if there is no row win,
   otherwise return the player that won. *)

   If[board[[1]] == board[[2]] == board[[3]] == "x" ||
     board[[4]] == board[[5]] == board[[6]] == "x" || board[[7]] == board[[8]] == board[[9]] == "x", 1,
    If[board[[1]] == board[[2]] == board[[3]] == "o" || board[[4]] == board[[5]] == board[[6]] == "o" ||
      board[[7]] == board[[8]] == board[[9]] == "o", 2, 0]]
  ];

checkColWin[board_] := Module[{},
   (* SPECIFICATION: Returns 0 if there is no column win,
   otherwise return the player that won. *)

   If[board[[1]] == board[[4]] == board[[7]] == "x" ||
     board[[2]] == board[[5]] == board[[8]] == "x" || board[[3]] == board[[6]] == board[[9]] == "x", 1,
    If[board[[1]] == board[[4]] == board[[7]] == "o" || board[[2]] == board[[5]] == board[[8]] == "o" ||
      board[[3]] == board[[6]] == board[[9]] == "o", 2, 0]]
  ];

checkDiagWin[board_] := Module[{},
   (* SPECIFICATION: Returns 0 if there is no diagonal win,
   otherwise return the player that won. *)

   If[board[[1]] == board[[5]] == board[[9]] == "x" || board[[3]] == board[[5]] == board[[7]] == "x", 1,
    If[board[[1]] == board[[5]] == board[[9]] == "o" || board[[3]] == board[[5]] == board[[7]] == "o", 2, 0]]
  ];

checkBoardStatus[board_] := Module[{rowWin, colWin, diagWin, result, i},
   (* SPECIFICATION: Returns -1 if the game is still in progress,
   0 if there is a draw, 1 if "x" wins, and 2 if "o" wins. *)

   (* Check if there is a win for either player. *)
   rowWin = checkRowWin[board];
   colWin = checkColWin[board];
   diagWin = checkDiagWin[board];
   result = If[rowWin > 0, rowWin, If[colWin > 0, colWin, If[diagWin > 0, diagWin, 0]]];
```

```
    (* If there is not a win, then check for any blank spaces. If there are blanks,
    the game is still in progress. If there are none, the game is a draw. *)
    If[ result == 0, For[i = 1, i ≤ 9, i++,
      If[board[[i]] == "b", {result = -1; i = 10;}, Null]], Null];

    result
   ];

simulatePlayOut[index_] :=
  Module[{currBoard, currPlayer, opponent, boardStatus, parentNodeIndex, w},
    (* SPECIFICATION: Play out a game from the node with
      the input index and return the final board status. *)

    currBoard = board /. n[index];
    currPlayer = player /. n[index];
    opponent = 3 - currPlayer;
    boardStatus = checkBoardStatus[currBoard];

    (* If the result from n[index] is a loss,
    then set the parent's win score as low as possible. Essentially,
    we are telling the current player to never go to the
     parent node of n[index] because it will lead to a loss. *)
    If[boardStatus == opponent,
     {
      parentNodeIndex = parent /. n[index];
      w = winCount /. n[parentNodeIndex];
      n[parentNodeIndex] = n[parentNodeIndex] /. (winCount → w) → (winCount → -1000000);
     },
     {
      (* While a game is incomplete, play out a game until the end. *)
      While[boardStatus == -1,
        {
         currBoard = RandomChoice[generateNextBoards[currBoard, currPlayer]];
         (* Add a move onto the board. *)
         boardStatus = checkBoardStatus[currBoard]; (* Check if the game has ended. *)
         currPlayer = 3 - currPlayer; (* Toggle the player. *)
        }
       ];
     }
    ];

    boardStatus
  ];
```

Backpropagation

In[•]:=
```
backPropagation[nodeToExploreIndex_, playoutResult_] :=
  Module[{currNodeIndex, v, vNew, w, wNew},
   (* SPECIFICATION: Updates the values for all the nodes
     from the exploration node to the top node. In this,
    the "top node" is the current game state from which a new move must be made. *)

   currNodeIndex = nodeToExploreIndex;

   While[currNodeIndex ≠ "none",
    {
     (* Increment the number of visits for the current node. *)
     v = visitCount /. n[currNodeIndex];
     vNew = v + 1;
     n[currNodeIndex] = n[currNodeIndex] /. (visitCount → v) → (visitCount → vNew);

     (* If the opponent of the player to make a move at the current
      node is the winner of the playout, increment the win count. *)
     If[3 - (player /. n[currNodeIndex]) == playoutResult,
      {
       w = winCount /. n[currNodeIndex];
       wNew = w + 1;
       n[currNodeIndex] = n[currNodeIndex] /. (winCount → w) → (winCount → wNew);
      },
      Null];

     (* Change the node whose values
      are being updated to the parent of the current node. *)
     currNodeIndex = parent /. n[currNodeIndex];
    }
   ];
  ];
```

A note on backpropagation:



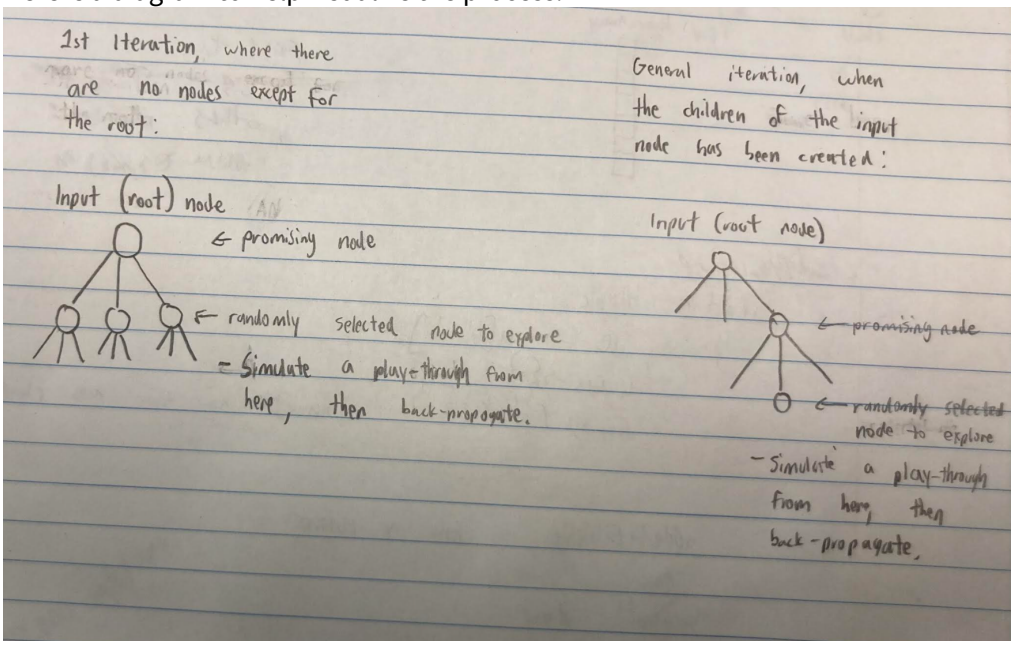
The simulation ended in Player 1 winning. As the algorithm backtracks, for the middle node, the win count increases. Since at the root node, Player 1 is to make the next move, the algorithm should more often choose the node that is on the path to a win (the middle node, in this case). The win count does not increase for the root node because Player 2 is choosing whether or not to move to the root node, and since it is on path to potentially losing, the UCB value should not increase.

Find Next Move Phase

The findNextMove function represents one step of the MCTS. It generates the next best move given the current state. In order to preserve memory, we are wiping the nodes after each step and re-generating them with the newest game state as the root of the tree. The performance values of the new nodes will be updated in the simulation and backpropagation phase. These new nodes will be used to determine the next move. Then, the nodes are wiped again.

One step has a for-loop with many iterations (1000 in this case). In each iteration:
1. Select the most promising node to explore, which will be input node itself in the first iteration (since all the nodes have been wiped and the algorithm has yet to generate its children) or the child of the input node with the highest UCB value in the other iterations.
2. Generate the children for the most promising node if the game is not yet complete and the most promising node does not have any children.
3. Choose a random child of the most promising node to explore and simulate a random play-through from here to the end.
4. Back-propagate the results of the play-through.

Here is a diagram to help visualize the process:



In[ ]:=
```
getBestChild[index_] :=
  Module[{childrenList, bestNodeIndex, max, currNodeIndex, currVisitCount, i},
```

```
    (* SPECIFICATION: From the node with the input index,
    pick the index of child node with the most visits. *)

    childrenList = children /. n[index];
    bestNodeIndex = First[childrenList];
    max = visitCount /. n[bestNodeIndex];

    For[i = 2, i ≤ Length[childrenList], i++,
     {
      currNodeIndex = childrenList[[i]];
      currVisitCount = visitCount /. n[currNodeIndex];
      If[currVisitCount > max,
       {max = currVisitCount; bestNodeIndex = currNodeIndex;}, Null];
     }
    ];

    bestNodeIndex
   ];

findNextMove[currBoard_, currPlayer_] := Module[{index, promisingNodeIndex, childrenList,
     nodeToExploreIndex, oldParentIndex, playoutResult, nextNodeIndex, i},
    (* SPECIFICATION: Returns the board after making a decision on
      which move to make. This represents one step of the MCTS. *)

    index = StringJoin[currBoard];

    For[i = 0, i < 1000, i++,
     {
      (* From the input node, select the child with the highest UCB
       value as the most promsing node. If the input node has no children,
      the input node becomes the most promising node. *)
      promisingNodeIndex = selectPromisingNode[index];
      childrenList = children /. n[promisingNodeIndex];

      (* If the most promising node represents a game
       in progress and it has no children, generate its children. Since
       the game is unifinished, there must be a future state. *)
      If[checkBoardStatus[board /. n[promisingNodeIndex]] == -1 &&
        Length[childrenList] == 0,
       {
        expand[promisingNodeIndex];
        childrenList = children /. n[promisingNodeIndex];
       },
       Null];

      (* If the most promising node has children,
      pick a random one to explore. Otherwise, explore the most promising node. *)
```

```
  If[Length[childrenList] > 0,
   {
    nodeToExploreIndex = RandomChoice[childrenList];
    (*
    oldParentIndex=parent/.n[nodeToExploreIndex];
    n[nodeToExploreIndex]=
     n[nodeToExploreIndex]/.(parent→oldParentIndex)→(parent→promisingNodeIndex);
    *)
   },
   nodeToExploreIndex = promisingNodeIndex];

  (* Play out a game to from the node to explore and back propogate the result. *)
  playoutResult = simulatePlayOut[nodeToExploreIndex];
  backPropagation[nodeToExploreIndex, playoutResult];
 }
];

nextNodeIndex = getBestChild[index];
board /. n[nextNodeIndex]
];
```

Two AI's using the MCTS to make decisions should always draw.

In[ ]:=
```
PrintBoard[gameboard_] := Module[{newboard, boardByRows},
  (* SPECIFICATION: Print out the board in matrix form for easier visualization. *)

  newboard = gameboard /. "b" → " ";
  boardByRows = {newboard[[1 ;; 3]], newboard[[4 ;; 6]], newboard[[7 ;; 9]]};
  Print[MatrixForm[boardByRows]];
 ]

(* Declare the root node, which represents the empty board at the start of the game. *)
n["bbbbbbbbb"] = {parent → "none", board → {"b", "b", "b", "b", "b", "b", "b", "b", "b"},
   children → {}, visitCount → 0, winCount → 0, player → 1};
gameboard = {"b", "b", "b", "b", "b", "b", "b", "b", "b"};
playerNo = 1;

For[i = 0, i < 9, i++, {
   gameboard = findNextMove[gameboard, playerNo];
   result = checkBoardStatus[gameboard];
   If[result ≠ -1, i = 9, Null];
   playerNo = 3 - playerNo;

   (* Here, we are updating the "root" of the tree to the current board
    state. The previous moves have already been made and cannot be changed,
   so we only look toward the future moves and making decisions for those. *)
   index = StringJoin[gameboard];
   tempNode = n[index];
   oldParent = parent /. tempNode;
   oldChildrenList = children /. tempNode;
   Clear[n];
   n[index] = {parent → "none", board → gameboard,
     children → {}, visitCount → 0, winCount → 0, player → playerNo};

   PrintBoard[gameboard];
  }];

If[result == 0, Print["Draw!"],
  If[result == 1, Print["Player 1 wins!"], Print["Player 2 Wins!"]]];
```

```
{{ , , },
 { , x, },
 { , , }}

{{o, , },
 { , x, },
 { , , }}

{{o, , x},
 { , x, },
 { , , }}

{{o, , x},
 { , x, },
 {o, , }}

{{o, , x},
 {x, x, },
 {o, , }}

{{o, , x},
 {x, x, o},
 {o, , }}

{{o, x, x},
 {x, x, o},
 {o, , }}

{{o, x, x},
 {x, x, o},
 {o, o, }}

{{o, x, x},
 {x, x, o},
 {o, o, x}}

Draw!
```

Conclusion

The MCTS is an effective method of AI decision-making for board games given that it has been trained adequately. It does not rely on any knowledge of the game and instead relies solely on previous experience to make decisions. It is also more memory efficient than alternatives like Min-Max search, which requires pre-generating all the board states (even for a game like tic-tac-toe, there are many possible game boards). However, a MCTS AI will fail to perform optimally if it has not been trained properly, and training a MCTS algorithm extensively requires substantial memory and computing power, especially for complex games like chess or go.

References: https://www.baeldung.com/java-monte-carlo-tree-search