# Monte-Carlo Search Tree Algorithm and its Tic-Tac-Toe Application

Yuze Liu
Mechanical Engineering Department
Florida State University

**Abstract**—To build a machine player that can beat human player in the game is always a fascinating thing. As the final project of the course Stochastic Computing which introduces the Monte-Carlo Methods, this project uses the Monte-Carlo Tree Search to build an Artificial Intelligence that can play Tic-Tac-Toe game with human player. Tic-Tac-Toe game is a classic game with simple rules and relatively fewer possibilities (compare with GO and Chess).It's a good start to learn and implement the Monte-Carlo Tree Search Algorithm. This paper will summarize all the related knowledge of building this program and also explain the important idea and details to make such algorithm become a real program that can be runned and played with.

**Index Terms**—Monte-Carlo Tree Search, Tic-Tac-Toe, Upper Confidence Bounds, Artificial Intelligence, MiniMax, Alpha-beta pruning, Game tree search

## 1 Introduction

Computer game-playing was one of the most interesting and fascinating inquiries into Artificial Intelligence. The study of computer game-playing can be tracked back to the first chess-playing machine proposed by Shannon in 1950[1]. The recently breakthrough in game-playing should be the AlphaGo which is developed by Google DeepMind in London to play the board game Go[2].In March 2016, it beat Lee Sedol in a five-game match, the first time a computer Go program has beaten a 9-dan professional without handicaps.AlphaGo's algorithm uses Monte-Carlo Tree Search to find its moves based on knowledge previously "learned" by machine learning, specifically by an artificial neural network (a deep learning kind) by extensive training, both from human and computer play.

In order to get the basic idea of the Game tree search, especially the Monte-Carlo Tree Search, a program is built to play a very famous game Tic-Tac-Toe. In this program, human player will play Tic-Tac-Toe game with computer player who is going to use Monte-Carlo Tree Search algorithm to find the best move.In order to show the computer is smart by using Monte-Carlo Tree Search algorithm, the goal of the game result is either the computer is win or the game is draw in the end. In this way, we will say the computer is unbeatable, and the implement of this algorithm is successful.

## 2 Background

### 2.1 Tic-Tac-Toe Game

Tic-Tac-Toe(also known as Noughts and crosses or Xs and Os) game is a famous classic board game. Two player will play on a board with 3x3 grid on the board, two players will make move alternatively until one of the player succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game.

There are several good natures for this game which makes it popular to be used in doing game tree search research. The first one is Tic-Tac-Toe is a zero-sum game. In game theory and economic theory. a zero-sum game is a mathematical representation, each player gain(or loss) of utility is exactly balanced by the losses (or gains) of the utility of the other players. Second, for the traditional Tic-Tac-Toe game, there are only 9 grids on the board, if a game-tree is built for this game, it won't grow dramatically, compared with Go and other board games. These two natures make this game is easy to implement with lots of game tree algorithm, not only the Monte-Carlo Tree Search Algorithm but also other algorithm like minimax and alpha-beta pruning. Actually, the simplicity of the game makes it hard to show the real advantage of Monte-Carlo Tree Search algorithm, but it also makes the algorithm clear and easy to understand.

## 3 Game Trees and Tree Search Algorithm

### 3.1 Game Trees

One commonly used technique in computer game-playing is game tree search. The board game can be represented with a game tree. Figure 1 shows the game tree built for the Tic-Tac-Toe game. The root of the tree represent the current state of the Game and player "X" is going to make the next move, each edge from the root represent a different single move and each single move will lead the game to a new state. Since there are five open squares in the current state(can be seen from the root node), then player "X" has five different ways to make his move, so there are five edges from the root node. Then player "O" and player "X" will play alternatively, until a leaf node has been reached, a leaf node represents a en-state of the game, either there is a winner in the game or a draw.

Game trees allow computers to reason the result and the consequence of the game. They clearly shows the currently state of the game and all the possible moves that can be made from the current move and the next state lead by those moves. Ideally, a computer can reason all the possible moves, simulate all the possible states and make the best move to win the game. And there comes to the different search algorithms to decide which move will be made by the computer. Some of the Algorithms have some similarities with each other, but the main idea behind the similarities is totally different. In the following section will introduce two algorithms that has some kind of similarity with Monte-Carlo Tree Search Algorithm, but

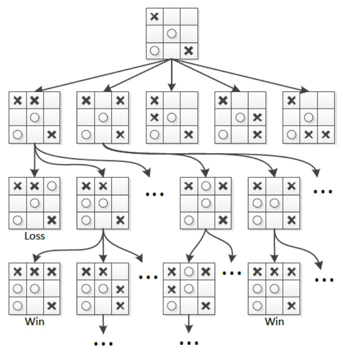

Fig. 1. A (partial) game tree for the game of Tic-Tac-Toe. The X-player is to act at the root node. Nodes represent positions in the game and edges represent moves.[3]

the core idea is totally different, study those two algorithms can actually help us to understand Monte-Carlo Tree Search better and those algorithms can also be hybrid and make new algorithms.

### 3.2 MiniMax

MiniMax algorithm is an algorithm is an algorithm for finding an optimal strategy in a certain game state for deterministic two-player zero-sum games.[3] In this algorithm, two players are marked with MAX-player(the player we want to find the best move for him) and MIN-player(the opposite player of the MAX-player),both players will be assumed to play optimally. The optimality implies that the MAX-player will always choose the move that will lead to the largest utility for him at last and oppositely the MIN-player will always find the move lead to the smallest utility for the MAX-Player. Each leaf node can be assigned a utility value which implies its utility perspective of the MAX-player. And from this utility value, we can assign each node in the tree with a MiniMAx value based on the following recursive function.

$$M(n) = \begin{cases} Utility(n) & \text{n} \in \text{ leaf node} \\ max_{s\in C(n)} M(s) & \text{n} \in \text{ max node} \\ min_{s\in C(n)} M(s) & \text{n} \in \text{ min node} \end{cases} \quad (1)$$

In this equation, M(n) stand for the Minimax value of the node n, C(n) stands for the

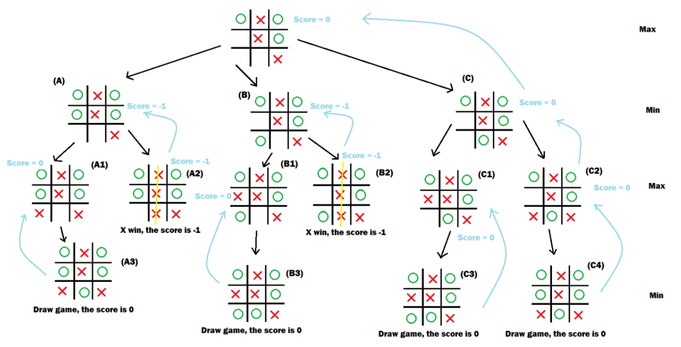

Fig. 2. A (partial) Minimax game tree for the game of Tic-Tac-Toe. The O-player is the MAX-Player, and X-player is the MIN-Player.

children of the node n, Utility(n) is the utility of leaf node n. This function can be used to write a Minimax algorithm which is a depth-first tree search algorithm.

Figure 2 illustrate the Minimax Algorithm. In this figure, the O-player is the MAX-Player and every move makes by the O-Plyaer is to choose the largest utility node and every move makes by the X-Player is to choose th smallest utility node. From the bottom leaf node back to the root, if the game ends with O-player wins the game, then the utility of that leaf node is 1, if it is a draw, the utility is 0, otherwise, it is -1. In this figure, the O-Player will choose the third move as his move since it's value is 0 which is largest among the three choices.

The Minimax algorithm is a very basic and only applies to deterministic games with 2 player. The main issue of this algorithm is it has to go through all the possible moves and then make the bes move. For Tic-Tac-Toe game, it works because the games is simple and the moves are very limited. If this algorithm is used for game like GO, then it's not very effective.

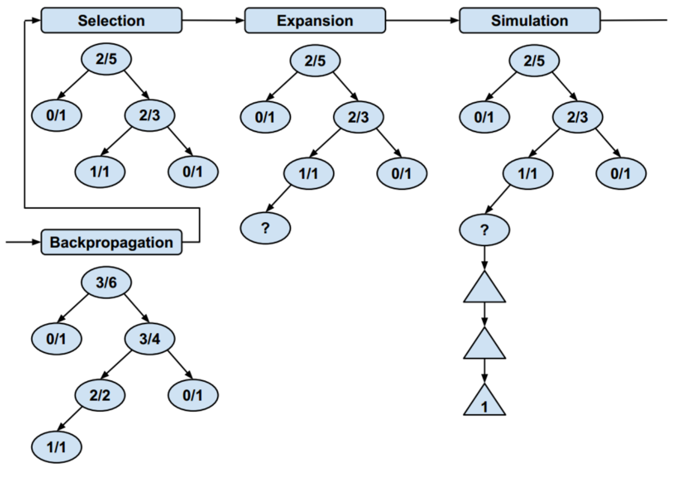

Fig. 3. Four Steps of the MCTS Algorithm[7]

## 4 Monte- Carlo Tree Search

Monte-Carlo methods are a class of alogrithim that rely on random sampling or simulations to approximate quantities that are very hard or even impossible to calculate exactly[3].Monte-Carlo Tree Search (MCTS)[4][5] is a best first tree search tree algorithm that evaluate each state by the average result of simulations from each state[6].Unlike Minimax algorithm, it doesn't need to go through all the possible moves, it samples moves and does random simulation to simualte the games. Thus it can handle large space tree search. MCTS works by repeating the following four-steps loop until it hits a boundary, the boundary can be a time limit, like do this loop for 5 minutes or it can also be the times that it will repeat, like do the loop 1000 times or any other boundaries. The four steps are: Selection, Expansion, Simulation and Backpropagation.

### 4.1 Selection

In the selection process, the algorithm will traverses the current tree from the root (current state of the game) by using a tree policy till it reaches a leaf node.A tree policy is actually an evaluation of the node by using some equation to calculate the value of the node. In the Figure 3, each number on the node represents "number of wins / number of the total game". So the tree policy of Figure 4 is to evaluate this number and choose the larger one. So from the root node, between 0/1 and 2/2, it chooses 2/3 by the tree policy, and then from 1/1 and 0/1 it chooses 1/1 since 1/1 is larger than 0/1. Once it reaches a leaf node of the current tree, it will do to the next step, Expansion.

### 4.2 Expansion

In the Expansion process, if the current leaf node which is reached by from the selection process is not a end-game state, which means it still has children that can be added, then a child node will be added to that node. From the figure, it is the node marked with question mark. From this node, it will go to the next step, Simualtion.

### 4.3 Simulation

In the Simulation process, the algorithm will start from the new-added node(marked with question mark in the graph) and do the simulation by using the default tree policy. During the simulation process, it will choose the moves until a end-game state or a predefined threshold is reached. In the case of Tic-Tac-Toe game, an end states is reached when the game is end. Then based on the simulation results, the value of the new-added node will be established. Still use Figure 4 as an example, the simulation results a win state, so the new-added node's value will be 1/1 which stands for it has been visited once (or does the simulation once) and it reaches a win state.
In this simulation process, there is a default policy mentioned above, the default policy may be either weak or strong. A weak policy will use little predetermined strategy. It just choose moves randomly from all the legal moves. To make it a strong policy, it can use some predetermined policy like if the middle square is empty, always put the mark at that position or if the corner position in more favorable, always choose that move.
Since the value of the new-added node is available now, the algorithm goes to the last step, Backpropagation.

### 4.4 Backpropagation

As the last step of the algorithm, it will update all the node's value from the new-added node to the root. If the node is the one that has been selected during the selection process, which means it has been used to do the simulation (in other words, it has been visited once) once, which will add 1 on each of the node's denominator. And since the simulation from the new-added node is a win state, so it will also add 1 on each of the node's numerator.

## 5 Upper Confidence Bound

The above statement mentioned tree policy during the selection process, to illustrate the basic idea of the algorithm , we only used the equation Wins/Total Games as the equation of the tree policy. But there is a more frequently used tree policy in MCTS algorithm which is the Upper Confidence Bound. This is also the tree policy that is used in the Tic-Tac-Toe game I built.
The upper confidence bound applied to trees (UCT) balance the idea of exploration versus exploitation.Exploration promotes exploring unexplored area in the tree. It will expand the tree's breadth more than its depth. Exploration will ensure that the algorithm will not overlook any potential optimal path of of the tree.Oppositely, exploitation will stick to one path that has the greatest estimated value(this value is defined by the tree policy). Exploitation is greedy and will extend the tree's depth more than breadth.
UCT balance the exploration and exploitation by using the equation:

$$UCT(node) = \frac{W(node)}{N(node)} + c\sqrt{\frac{ln(N(parentNode))}{N(node)}} \tag{2}$$

In this equation, W(node) represents how many winning result in all of those simulations of that node and N(node) represents how many simulations of that node. The $\frac{W(node)}{N(node)}$ contributes the Exploitation of the UCT value. The more it wins from this node, the algorithm will more prefer to choose this node and the path follows this node, so it prefers to choose the path that has been visited and go deeper and deeper. In the second part of the equation, it consider the number that the node has been visited as well as the number the parent node of this node has been visited. If the node has been visited many times, the denominator of the second part will become larger and the whole value will become smaller. The UCT value of the node that has been visited more will be smaller, so the algorithm will prefer to choose the node that has been visited less, in this way, the second part of the equation will help to contribute the exploration of the algorithm. c is the constant choose by people to balance the relationship between the exploration and the exploitation. Usually, we choose $\sqrt{2}$

## 6 Using MCTS to Plat Tic-Tac-Toe

As it mentioned in the previous part, the main part of this project is to build an actual AI that can play Tic-Tac-Toe game with human player. Fo this project, a program has been built by using Visual Studio 2013 and C++ language. The pseudo code of the program is:

Algorithm 1 Pseudo Code of the UCT[8]

```
Function UctSearch($s_0$)
  create root node $v_0$ with state $s_0$
  while in the computational budget do
    $v_{leaf} \leftarrow Treepolicy(v_0)$
    simulation result $\leftarrow Defaultpolicy(s(v_l))$
    $Backup(v_l,$ simulation result $)$
  return $a(BestChild(v_0, 0))$

Function Treepolicy($v$)
  while v is nonterminal do
    if $v$ is not fully expanded then
      return Expand($v$)
    else
      $v \leftarrow Bestchild(v_0, c)$
  return $v$

Function Expand($v$)
  choose a $\in$ untried actions from A($s(v)$)
  add a new child $v'$ to $v$
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return $v'$

Function BestChild($v, c$)
  return $argmax_{v' \in \text{children of} v} \frac{Q(v')}{N(v')} + c\sqrt{\frac{ln(N(v))}{N(v')}}$

Function DefaultPolicy($s$)
  while s is nonteminal do
    choose a$\in$ A(s) uniformaly at random
    $s \leftarrow f(s, a)$
  return reward for state $s$

Function Backup($v$, simulation result)
  while v is not null do
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) +$ smulation result
    $v \leftarrow$ parent of $v$
```

The pseudo code shows the basic idea that how to implement the UCT algorithm to the game. In the actual program of the game, there are actually slightly difference from the classic UCT Algorithm.
The actual UCT equation actually is:
$UCT(node) = \frac{W(node)}{N(node)+\epsilon} + c\sqrt{\frac{ln(N(parentNode))}{N(node)+\epsilon}}$

$$UCT(node) = UCT(node) + random() * \epsilon \quad (3)$$

The $\epsilon$ in the equation is to avoid program error. In the beginning, the value of N(node) is 0 which will cause error when run the program. The sceond equation is used to add randomization since during the Expand step, instead of only add one node, we will add all the possible chid node of the last node from the selection step. The UCT value of all the chidnode might be the same without the second equation, then in order to choose the chid node to do the simulation randomly, we add the second equation of to add randomization.
Also, the boundary condition of the algorithm is it will not stop until it repeats 15000 times.
Another thing worth to mention is instead of making the best move based on the MCTS algorithm, another condition has been applied to the algorithm as well as the default policy in the simulation step to strengthen the default policy as I mentioned previously in the 4.3. There is another function called Block, at the AI's turn, when the AI detect there exists a immediately move that can lead the human player win the game, it will directly make that move to block the human player. This also make the AI more smart.

## 7 Running Result and Conclusion

The running result is promising, the AI we designed reached the goal that it is unbeatable. As far as I tested, the AI can keep winning the game or lead the game to a draw state. There may exist some trap that is not detected yet, but so far the AI acts very smartly.

There does exist one deflection of the program I found out recently. Sometimes The AI will not use a smart move to win the game immediately, for example, when there exists a move that can make the AI win the game, and in human player's turn, the human player makes another move that does't block the potential winning move of the AI but this move is also a move that has the potentiality to make the human win the game When it comes to the AI's turn, if the computer makes the move that can make itself win immediately,the game will end, but the actual running result shows that the AI will choose to block human player's move instead of making itself win. This is mainly caused by the Block function and can be fixed by add another function to force the computer makes the move that will win the game or can be fixed by deleting the block function and let the MCTS Algorithm make the choice all by itself.

## Appendix A
## Code of the Program

The actual code I wrote for the game will add at the very end of the paper. Since it may look discontinuous if I add all the code directly in the text of the paper.

## References

[1] C.E.Shannon, *"Programming a computer to play chess," Philosophy Magazine*, Vol. 41, 1950, pp. 256-275

[2] *"Artificial intelligence: Google's AlphaGo beats Go master Lee Se-dol"*. BBC News. Retrieved 17 March 2016.

[3] A.A.J. van der Kleij, *"Monte Carlo Tree Search and Opponent Modeling through Player Clustering in no-limit Texas Hold'em Poker"*, master thesis

[4] R. Coulom, *Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search,* in 5th International Conference on Computers and Games (CG 2006). Revised Papers, ser. Lecture Notes in Computer Science, H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, Eds., vol. 4630. Springer, 2007, pp. 7283.

[5] L. Kocsis and C. Szepesvari, *Bandit Based Monte-Carlo Planning,* in 17th European Conference on Machine Learning, ECML 2006, ser. Lecture Notes in Computer Science, J. Furnkranz, T. Scheffer, and M. Spiliopoulou, Eds., vol. 4212. Springer, 2006, pp. 282293.

[6] Hendrik Baier and mark H.M. Winands, *Monte-carlo Tree Search and Minimax Hybrids* 2013 IEEE

[7] Max Magnuson*Monte-carlo Tree Search and Its Application* University of Minnesota morris Digital Well, 2015

[8] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton*A Survey of Monte Carlo Tree Search Methods* IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, VOL. 4, NO. 1, MARCH 2012

## //Appendix A

```
//*********** Monte-Carlo-Search-Tree Tic-Tac-Toe Game ************//
//*                     Yuze Liu                              *//
//*               ME Department - FSU                          *//
//*****************************************************************//
#include<iostream>
#include<cstdio>
#include<vector>
#include<string>
#include<math.h>
#include<algorithm>
#include<random>
#include<ctime>
#include<thread>
#include<chrono>
#include<omp.h>
using namespace std;


class Node
{
    int w;      //Number of wins
    int v;      //Number of visits;
    Node* parent;
    vector<Node*>children;
    vector<char>board;
    char currentSymbol;
    int pos;
public:
    Node()
    {
        for (int i = 0; i < 9; i++)
        {
            board.push_back('-');
        }
        parent = NULL;
        w = 0;
        v = 0;
    }
    ~Node()
    {
        parent = NULL;
        children.clear();
        board.clear();
    }
    Node(vector<char>board, Node* parent)
    {
        w = v = 0;
        this->board = board;
        this->parent = parent;
    }

    int getMoves(){ return pos; }
    void setMoves(int value){ pos = value; }
    vector<char>getBoard(){ return board; }
    void setCurrentSymbol(char value){ currentSymbol = value; }
    char getCurrentSymbol(){ return currentSymbol; }
    Node* getParent(){ return parent; }
    vector<Node*>getChildren(){ return children; }
    int getNoOfWins(){ return w; }
    int getNoOfVisits(){ return v; }
    bool isEndState();
    bool isBoardEmpty();
    int getNoOfValidMoves(){ return children.size(); }
    bool isLeaf();
    void print_moves();
    Node* select();
    void expand();
    int simulate(Node* n, char opponent_symbol);
    void update(int value);
    int genRandomNumber();
};



int Node::genRandomNumber()
{
    int lo = 0;
    int high = 8;
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<int>rand_gen(lo, high);

    return rand_gen(rd);
}
```

```cpp
bool Node::isLeaf()
{
    return (children.size() == 0);
}

void Node::print_moves()
{
    int c = 0;
    for (int i = 0; i < 9; i++)
    {
        cout << board[i] << " ";
        c++;
        if (c % 3 == 0)cout << endl;
    }
    cout << endl;
}


bool Node::isBoardEmpty()
{
    bool isEmpty = false;
    vector<char>temp_board = this->getBoard();
    for (int i = 0; i < 9; i++)
    {
        if (temp_board[i] == '-')
        {
            isEmpty = true;
        }
        else
            isEmpty = false;
    }

    return isEmpty;
}

bool Node::isEndState()
{
    bool endState = true;
    for (int i = 0; i < 9; i++)
    {
        if (board[i] == '-')
        {
            endState = false;
        }
    }

    return endState;
}

bool checkEndState(vector<char>board)
{
    bool end = true;
    for (int i = 0; i < 9; i++)
    {
        if (board[i] == '-')
        {
            end = false;
        }
    }
    return end;
}

bool checkWinState(vector<char>board, char symbol)
{
    int pos[8][3] = {
        { 0, 1, 2 },
        { 3, 4, 5 },
        { 6, 7, 8 },
        { 0, 3, 6 },
        { 1, 4, 7 },
        { 2, 5, 8 },
        { 0, 4, 8 },
        { 2, 4, 6 }
    };
    int c = 0;
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (board[pos[i][j]] == symbol)
            {
                c++;
            }
        }
        if (c == 3)
        {
            return true;
        }
        c = 0;
    }

    return false;
}

bool checkDrawState(vector<char>board, char symbol)
{
    bool draw = false;
    if (checkEndState(board))
    {
        int pos[8][3] = {
            { 0, 1, 2 },
            { 3, 4, 5 },
            { 6, 7, 8 },
            { 0, 3, 6 },
            { 1, 4, 7 },
            { 2, 5, 8 },
            { 0, 4, 8 },
            { 2, 4, 6 }
        };
        int c = 0;
        for (int i = 0; i < 8; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                if (board[pos[i][j]] == symbol)
                {
                    c++;
                }
            }
            if (c < 3)
            {
                draw = true;
            }
            else
            {
                draw = false;
            }
            c = 0;
        }
    }
    return draw;
}
```

```
int check(vector<char>board, int& p, char current_symbol)
{
    int c = 0;
    int b = 0;
    int pos[8][3] = {
        { 0, 1, 2 },
        { 3, 4, 5 },
        { 6, 7, 8 },
        { 0, 3, 6 },
        { 1, 4, 7 },
        { 2, 5, 8 },
        { 0, 4, 8 },
        { 2, 4, 6 }
    };
    int row = 0;
    int col = 0;
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (board[pos[i][j]] == current_symbol)
            {
                c++;
            }
            else if (board[pos[i][j]] == '-')
            {
                b++;
                row = i;
                col = j;
            }
        }
        if (c == 2 && b == 1)
        {
            p = pos[row][col];
            return p;
        }
        c = 0;
        b = 0;
    }
    return -1;
}

// This function is to check, if there is an inmediate win for the player, the
```

```cpp
computer will block that empty position.
int block(vector<char>board, int& p, char opponent_symbol)
{
    int o = 0;
    int b = 0;
    int pos[8][3] = {
        { 0, 1, 2 },
        { 3, 4, 5 },
        { 6, 7, 8 },
        { 0, 3, 6 },
        { 1, 4, 7 },
        { 2, 5, 8 },
        { 0, 4, 8 },
        { 2, 4, 6 }
    };
    int r = 0;
    int c = 0;
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (board[pos[i][j]] == opponent_symbol)
            {
                o++;
            }
            else if (board[pos[i][j]] == '-')
            {
                b++;
                r = i;
                c = j;
            }
        }
        if (o == 2 && b == 1)
        {
            p = pos[r][c];
            return pos[r][c];
        }
        o = 0;
        b = 0;
    }
    return -1;
}

//Monte Carlo Methods
```

```
Node* Node::select()
{
    double epsilon = 1e-6;
    double uctValue = 0.0;
    double bestValue = -numeric_limits<double>::max();
    Node* selectedChild = nullptr;
    for (auto child : children)
    {
        // Calculate the UCT values, apply UCT as the tree selection rule
        uctValue = (double)((child->getNoOfWins() / (child->getNoOfVisits() +
epsilon)) + (double)sqrt(log(getNoOfVisits() + 1) / (child->getNoOfVisits() +
epsilon)));
        uctValue += rand()*epsilon;
        //Without the above line, durig the select procedure, if two chidren
have the same value, it will always choose the one that it saw first.
        if (bestValue < uctValue)
        {
            bestValue = uctValue;
            selectedChild = child;
        }
    }
    return selectedChild;
}

void Node::expand() // Instead of expanding only one child from the leaf node,
it will expand all possible children of the leaf node.
{
    if (!this->isLeaf())return;
    vector<char>t_board = this->getBoard();
    for (int i = 0; i < 9; i++)
    {
        if (t_board[i] == '-')
        {
            t_board[i] = this->getCurrentSymbol();
            Node* temp = new Node(t_board, this);
            temp->setCurrentSymbol(this->getCurrentSymbol());
            temp->setMoves(i);
            children.push_back(temp);
            t_board[i] = '-';
        }
    }
}

int Node::simulate(Node* n, char opponent_symbol)
```

```
{
    vector<char>board_t = n->getBoard();
    srand(time(0));
    bool myTurn = false;

    while (!checkEndState(board_t))
    {
        int _pos;
        if (myTurn)
        {
            if (block(board_t, _pos, opponent_symbol) != -1)
            {
                _pos = block(board_t, _pos, opponent_symbol);
            }
            else if (check(board_t, _pos, currentSymbol) != -1)
            {
                _pos = check(board_t, _pos, currentSymbol);
            }
            else
            {
                _pos = rand() % 9;
                while (board_t[_pos] != '-')
                {
                    _pos = rand() % 9;
                }
            }
            board_t[_pos] = currentSymbol;
            if (checkWinState(board_t, currentSymbol))return 1;
            myTurn = false;
        }
        else if (!myTurn)
        {
            int _pos = rand() % 9;
            while (board_t[_pos] != '-')
            {
                _pos = rand() % 9;
            }
            board_t[_pos] = opponent_symbol;
            if (checkWinState(board_t, opponent_symbol))return -1;
            myTurn = true;
        }

    }
    return 0;
```

```
}

void Node::update(int value) // Backpropagation
{
    this->v++;
    this->w += value;
}


//player class will execute monte carlo framework, it will follow the MCST
algorithm.
class Player
{
    vector<char>board;
    bool isMyTurn = false;
    char currentSymbol;
    char opponentSymbol;
    int move;
public:
    Player(vector<char>board, bool isMyTurn, char currentSymbol, char
opponentSymbol)
    {
        this->board = board;
        this->isMyTurn = isMyTurn;
        this->currentSymbol = currentSymbol;
        this->opponentSymbol = opponentSymbol;
    }

    vector<char>getBoard(){ return board; }
    char getCurrentSymbol(){ return currentSymbol; }
    char getOpponentSymbol(){ return opponentSymbol; }
    void play(int n);
    int getBestMove(){ return move; }
};
//Framework for select ,expand ,playout and update these four task will repeat
to the given number of time
void Player::play(int n)
{
    int value;
    Node* temp = NULL;
    Node* root = new Node(board, NULL);

    for (int i = 0; i < n; i++)
    {
        Node* curr = root;
        curr->setCurrentSymbol(currentSymbol);
        while (!curr->isLeaf())
        {
            curr = curr->select();
        }

        if (curr->isLeaf() && curr->isEndState())
        {
            if (checkWinState(curr->getBoard(), currentSymbol))
            {
                value = 1;
            }
            else if (checkWinState(curr->getBoard(), opponentSymbol))
            {
                value = -1;
            }
            else
            {
                value = 0;
            }
        }
        else
        {
            curr->expand();
            Node* node = curr->select(); // it will only select one of the
child to simulate from all the possible children.
            value = curr->simulate(node, opponentSymbol);
            curr = node;
        }
        while (curr->getParent() != NULL)
        {
            curr->update(value);
            curr = curr->getParent();
        }
        curr->update(value);
    }

    vector<Node*>children = root->getChildren();
    auto itr = max_element(children.begin(), children.end(), [](Node* n1, Node*
n2)->
        bool{return n1->getNoOfVisits() < n2->getNoOfVisits(); });
    Node* best = *itr;
    move = best->getMoves();
    best->print_moves();
    board = best->getBoard();
    isMyTurn = false;
    children.clear();
}


class HumanPlayer
{

    vector<char>board;
    char currentSymbol;
    char opponent_symbol;
public:
    HumanPlayer(vector<char>board, char currentSymbol, char opponent_symbol)
    {
        this->board = board;
        this->currentSymbol = currentSymbol;
        this->opponent_symbol = opponent_symbol;
    }

    vector<char>getBoard(){ return board; }
    void setMove();
    char getCurrentSymbol(){ return currentSymbol; }
    char getOpponentSymbol(){ return opponent_symbol; }
    void PrintBoard();
};


void HumanPlayer::PrintBoard()
{
    int c = 0;
    for (int i = 0; i < 9; i++)
    {
        cout << board[i] << " ";
        c++;
        if (c % 3 == 0)cout << endl;
    }
    cout << endl;
}


void HumanPlayer::setMove()
{
    cout << "Your Turn!" << endl;
    int pos;
label: cin >> pos;
    if (board[pos] == '-')
    {
        board[pos] = currentSymbol;
    }
    else
    {
        cout << "Invalid Move!!" << endl;
        goto label;
    }
}

//------------------------------------------------------------------------------
------------------------------//

int bot1_w = 0;
int bot2_w = 0;
int draw = 0;
void Game()
{
    Node* root = new Node();
    vector<char>board = root->getBoard();

    while (!checkEndState(board))
    {
        cout << "Bot 1" << endl;
        Player* bot1 = new Player(board, true, 'x', 'o');
        bot1->play(15000);
        board = bot1->getBoard();
        if (checkWinState(board, bot1->getCurrentSymbol()))
        {
            cout << "Bot 1 wins!!!" << endl;
            bot1_w++;
            break;
        }
        if (checkDrawState(board, bot1->getCurrentSymbol()))
        {
            cout << "Draw" << endl;
            draw++;
            break;
        }
```

```cpp
        cout << "Bot 2" << endl;
        Player* bot2 = new Player(board, true, 'o', 'x');
        bot2->play(15000);
        board = bot2->getBoard();
        if (checkWinState(board, bot2->getCurrentSymbol()))
        {
            cout << "Bot 2 wins!!!" << endl;
            bot2_w++;
            break;
        }
        if (checkDrawState(board, bot2->getCurrentSymbol()))
        {
            cout << "Draw" << endl;
            draw++;
            break;
        }
        this_thread::sleep_for(chrono::seconds(3));
    }
}

int computerWin = 0;//*
int computerTie = 0; //*

void Game2()
{
    Node* root = new Node();
    vector<char>board = root->getBoard();
    while (!checkEndState(board))
    {
        cout << "Bot 2" << endl;
        Player* bot2 = new Player(board, true, 'o', 'x');
        bot2->play(15000);
        board = bot2->getBoard();
        if (checkWinState(board, bot2->getCurrentSymbol()))
        {
            cout << "Bot 2 wins!!!" << endl;
            computerWin++; //*
            break;
        }

        if (checkDrawState(board, bot2->getCurrentSymbol()))
        {
            cout << "Draw" << endl;
            computerTie++; //*
```

```cpp
                break;
            }


            HumanPlayer* hp = new HumanPlayer(board, 'x', 'o');
            hp->setMove();
            hp->PrintBoard();
            board = hp->getBoard();
            if (checkWinState(board, hp->getCurrentSymbol()))
            {
                cout << "You won!!!" << endl;
                break;
            }
            if (checkDrawState(board, hp->getCurrentSymbol()))
            {
                cout << "Draw" << endl;
                break;
            }

        }
}


int main()
{
    int matches = 20;
    int rounds = 0;
    computerWin = 0; //*
    computerTie = 0; //*

    for (int i = 0; i < matches; ++i)
    {
        cout << "-----------------------------ROUND " << i + 1 << "------------
-------------------" << endl;
        Game2();
        rounds++;
        cout << "Computer wins ration : " << computerWin << " / " << rounds <<
" = " << computerWin / rounds << endl;
        cout << "Computer ties ration : " << computerTie << " / " << rounds <<
" = " << computerTie / rounds << endl;
    }

    return 0;
}
```