Description

(a) Brief introduction of our method

We decide to use Monte Carlo Tree Search (MCTS) method to solve this tic-tac-toe problem. Although tic-tac-toe is a fairly small game which can be solved with an exhaustive search method, with respect to future extension, MCTS could be a better choice. MCTS is based on random sampling of game states, meanwhile it can well balance exploration and exploitation. This property makes MCTS capable of relatively large game. Besides, tic-tac-toe is a combinatorial game (zero-sum, perfect information, deterministic, discrete, sequential), and MCTS suits this kind of game very well. The MCTS method mainly contains four steps, selection, expansion, simulation and backpropagation. We will briefly describe these fours steps together with our implementation.

Create a game board

Firstly, we define a game board which is specific for the tic-tac-toe game. The board is treated as a 3*3 matrix. To better calculate the result, the number 8 stands for 'O' and number 7 stands for 'X'. Whenever either of the columns, rows or diagonals add up to 24 or 21, one of the players wins and the game is terminated.

Create a node class

The 'Node' class is to track the current node in the search tree and store global variables incl. parent, child, visited times and winning times for UCB calculation (Upper Confidence bounds for Trees).

Main MCTS function

This function realizes the MCTS iterations. Each iteration includes the four steps of selection, expansion, simulation and backpropagation. The iteration time is depending on the time resource and computer resource. The more iterations, the more samplings are made, which means the USB value of each node becomes more and more accurate. In this case, machine can make better decision for the next step. After the iterations, a node is selected for the next step based on the winning rate.

```
# Main function of MC algorithm
def monte_carlo_tree_search(rootstate, iteration):
    root = Node(board=rootstate)
    for i in range(iteration):
        node = root
        board = rootstate.copy()
        node, board = tree_policy(node, board)
        board = default_policy(node, board)
        backup(node, board)
    s = sorted(root.children, key=lambda c:c.wins/c.visits)
```

Selection – tree_policy method

This method realizes selection and expansion in the MCTS function. The method will check if there is any child node which has not been explored yet.

- If so, it will randomly pick an unexplored child node and **expand** to that new node. This parent/child relationship will be stored in the node class variable. (step a)
- If all the child nodes have been explored, then it will **select** a child with the highest UCB score and move to that node. This step will keep iterating until there are some unexplored child nodes below the current node. Then it goes to (step a) and expand a new node.

```
# Search and expand function
def tree_policy(node, board):
    # if fully expanded and not leaf node, find the best child
    while node.unexplored == [] and node.children != []:
        node = node.best_child()
        board.move(node.action)
    # if leaf, expand to new node
    if node.unexplored != []:
        a = random.choice(node.unexplored)
        board.move(a)
        node = node.expand(a, board.copy())
    return node, board
```

Simulation - default_policy method

This is the roll-out policy of the algorithm. Starting from the node we've just expanded, the algorithm will check the condition if there is still vacancy on the board and the game is not terminated. If so, it will randomly select a vacant position, place a chess and update the board state. This will run iteratively until the condition becomes False. In this way, a sampling is made.

```
# Simulate function
def default_policy(node, board):
    while board.vacancy_on_board() != [] and not board.result():
        board.move(random.choice(board.vacancy_on_board()))
    return board
```

Backpropagation – backup method

After the simulation step, the result as well as the visited times will be added to all the nodes from the root to the expanded node. If the result of simulation is winning for the current player, we add 1 to the win times, otherwise -1 if the other player is winning. If the result is draw, we add 0 to the win times. And we add 1 to visited times of all nodes through the path. This can be considered as an accumulation step from the sampling.

```
# Backpropagation function
def backup(node, board):
    while node is not None:
        result = board.result()
        if result:
            if node.board.player==board.player:
                result = 1
            else: result = -1
        else: result = 0
        node.update(result)
        node = node.parent
```

Find best child – best_child method

This method simply calculates the UCB values of all the child nodes and select the highest one. We follow the formula as $\frac{w_i}{n_i} + C\sqrt{\frac{Int}{n_i}}$, $\frac{w_i}{n_i}$ is the average value from the sampling which controls exploitation, and $C\sqrt{\frac{Int}{n_i}}$ indicates how often the mode is visited which controls exploration.

```
def best_child(self):
    s = sorted(self.children, key=lambda c:c.wins/c.visits+0.2*sqrt(2*log(self.visits)/c.visits))
    return s[-1]
```

Final game playing

As the final illustration of the implementation, human vs. machine. The computer reads our input as the move, and then machine places the next move. The game will continue until anyone of us wins or we reach a draw. It turns out that it's hard for us to win the machine. Most of the times we reached a draw and sometimes machine won.

```
b = Board()
# Play with the computer. Me as 8, computer as 7
while b.vacancy_on_board() != [] and not b.is_terminal():
    line = input("It's your turn: ")
    my_turn = line[1:len(line)-1]
    my_turn = my_turn.split(',')
    my_turn = [int(i) for i in my_turn]
    b.move(my_turn)
    machine_turn = monte_carlo_tree_search(b, 1000)
    b.move(machine_turn)
    print(b.state)
```

```
It's your turn: [1,1]
[[7. 0. 0.]
 [0. 8. 0.]
 [0. 0. 0.]]
It's your turn: [0,1]
[[7. 8. 0.]
 [0. 8. 0.]
 [0. 7. 0.]]
It's your turn: [1,0]
[[7. 8. 0.]
 [8. 8. 7.]
 [0. 7. 0.]]
It's your turn: [2,0]
[[7. 8. 7.]
 [8. 8. 7.]
 [8. 7. 0.]]
It's your turn: [2,2]
Draw
```

(b) Evaluate on our algorithm

There are many advantages of our algorithm based on MCTS. Firstly, it is computation efficient. In our implementation, we set the iteration times to 1000. Even with this big number of times, the machine can still make a right decision very quickly. We can of course increase/decrease the iteration times to balance the decision accuracy and the computer/time resource we have. Secondly, the algorithm can be easily adapted to larger games because it does not waste computation on all possible nodes. We have tried to expand the original game to a 5*5 grid. The response time of machine is just a little bit slower than the 3*3 grid. Besides, the algorithm can even be reused by different games. We just need to replace the game board with the new one. Thirdly, the learned policy of the algorithm is competitive. We have tested the system for many times, and the final results were always machine win or draw. We believe there is only a very small probability that human could beat the machine.

However, there are also some drawbacks of the MCTS algorithm. Because of the sampling property of MCTS method, it needs many iterations to get enough samples for an expert decision. Although on a 3*3 or 5*5 board, the sampling time is acceptable for 1000 iterations as in our implementation, a much bigger game board may cause some efficiency problem. In this case, we need to balance the trade-off between a fast decision or a more expert level decision. To make fast decision, the algorithm focuses more on exploitation. To make more expert level decision, on the other hand, the algorithm focuses more on exploration. In this sense, this can also be considered as a trade-off between exploration and exploitation.