Homework2

Searching algorithms for optimal decision-making in game theory and AI

DEADLINE 25th of November 23:59

1 Task 1 - 2p

Given a larger board, where $width = 7$ and $height = 6$, define the `class ConnectFour` which is a version of tic-tac-toe implemented before.

Hint: `class ConnectFour` inherits `class TicTacToe` and has an additional function which constraints the player to operate only in the lowest empty square of a column.

For further references Play Connect Four online and read more details here.

2 Task 2 - 5p

Implement the algorithm *Upper Confidence Bound for Trees* from the Monte Carlo Tree Search Algorithm family.

One iteration of the general **MCTS** algorithm is structured in four phases:

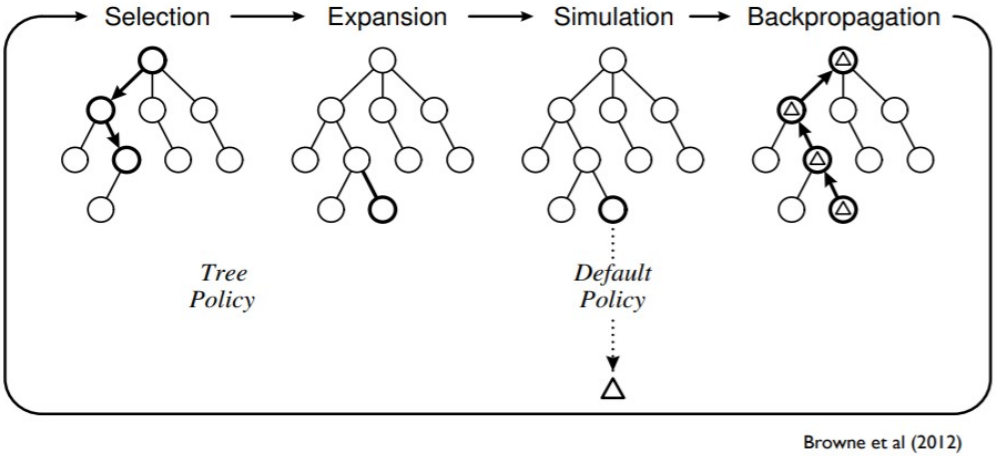


Figure 1: Four phases of the general MCTS algorithm, *Cameron Browne,* 2010

- **selection:** a strategy for selecting an action to exploit (choosing a leaf node in the tree)
- **expansion:** building a new node in the tree (append children's states to the leaf node)

- **simulation:** randomly pick a step and assume the utility of current state
- **back-propagation:** update the utility function for the parent nodes.

Algorithm 1 General MCTS approach.

```
function MCTSSEARCH(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TREEPOLICY(v_0)
        Δ ← DEFAULTPOLICY(s(v_l))
        BACKUP(v_l, Δ)
    return a(BESTCHILD(v_0))
```

Figure 2: The pseudocode for the MCTS algorithm, *Cameron Browne,* 2010

Start your research using this research paper.
Look for the action which maximises the following formula:

$$UCB1 = \frac{Q}{N} + c * \sqrt{\frac{2 * log(N_{node})}{N}} \tag{1}$$

Representing a node in MCTS as a class:

- the number of visits, N, representing the number of simulations performed from a node or a descendant of it.
- the estimated value Q, indicating the quality of the node, based on the number of games won starting from that node.
- N_{node} a reference to the parent node.
- children's list, a dictionary that contains for each action a link to the next node.
- c is a constant in the exploration and can be adjusted and can have a default value of $\frac{1}{\sqrt{2}}$.

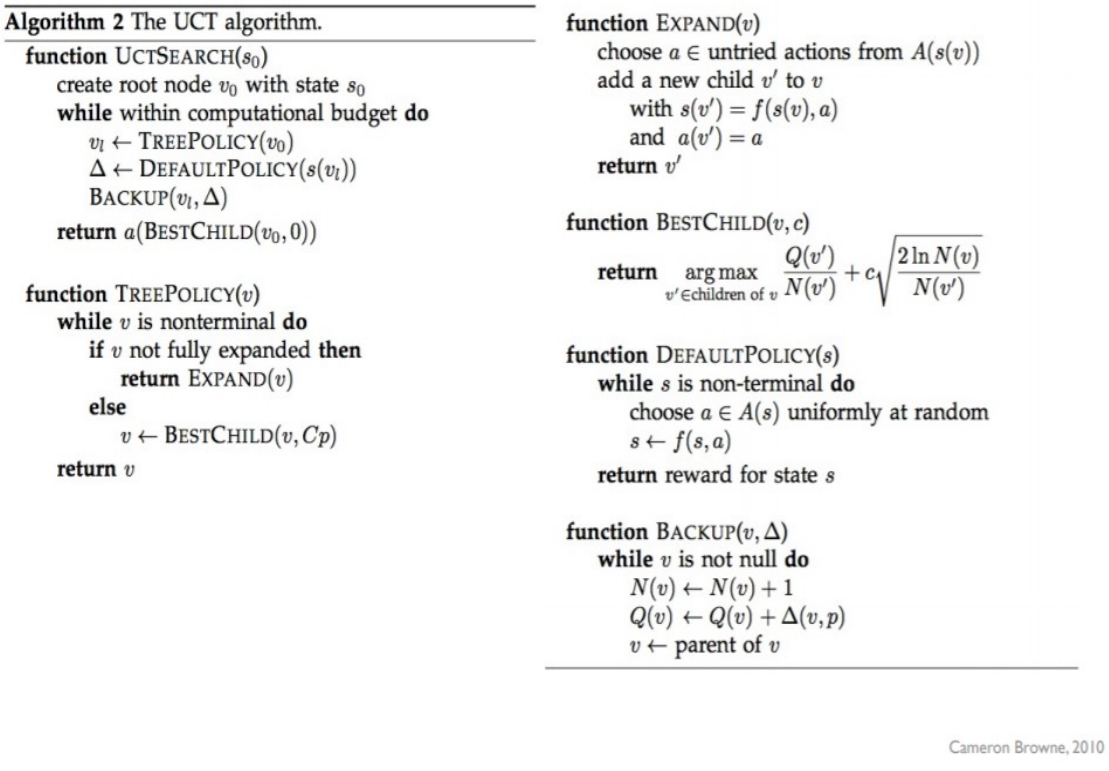
Evolution of the algorithm:

1. If the algorithm starts with an empty tree (no memory), then a new node is built. Otherwise, select the subtree according to the opponent's last action.

2. Until reaching the calculation budget limit:

- Starting from the root node, choose the next nodes successively until reaching a final state or a node from which not all possible actions have been explored.
- For a node that is not final and from which not all actions have been explored, build a child node for one of the unexplored actions.

- Simulate a game from the new node to a final state.
- Assess the final status and compute a reward is calculated.
- That reward spreads back, updating the statistics (number of visits) for each node to the root.



Algorithm 2 The UCT algorithm.

function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  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 $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

Cameron Browne, 2010

Figure 3: UCT pseudocode, *Cameron Browne,* 2010

The function call should look as follows:

```
play_game(ConnectFour(), dict(X=random_player, O=player(monte_carlo_tree_search))).utility
```

To define the actions of the problem, we use the *play_game* function receiving the current game to play and a strategy. The strategy itself reduces to a dictionary with the following structure:

```
{player_as_key : strategy_function}
```

where $strategy\_function$ can be called by: $strategy\_function(state, game)$

Example:

```
X: random_player -> random_player(state, game)
O: player(monte_carlo_tree_search) -> monte_carlo_tree_search(game, state)[1]
```

One possible output scenario is presented on the next page:

```
Player X move: (4, 5)
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . X . .
```

```
Player O move: (6, 4)
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . O . O
. X . . X O X
```

```
Player O move: (2, 4)
. . . . . .
. . . . . .
. . . . . .
. . . . X . .
. . O . O X O
. X O . X O X
```

```
Player O move: (5, 5)
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . X O .
```

```
Player X move: (4, 3)
. . . . . .
. . . . . .
. . . . . .
. . . . X . .
. . . . O . O
. X . . X O X
```

```
Player X move: (5, 3)
. . . . . .
. . . . . .
. . . . . .
. . . . X X .
. . O . O X O
. X O . X O X
```

```
Player X move: (6, 5)
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . X O X
```

```
Player O move: (2, 5)
. . . . . .
. . . . . .
. . . . . .
. . . . X . .
. . . . O . O
. X O . X O X
```

```
Player O move: (2, 3)
. . . . . .
. . . . . .
. . . . . .
. . O . X X .
. . O . O X O
. X O . X O X
```

```
Player O move: (4, 4)
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . O . .
. . . . X O X
```

```
Player X move: (5, 4)
. . . . . .
. . . . . .
. . . . . .
. . . . X . .
. . . . O X O
. X O . X O X
```

```
Player X move: (4, 2)
. . . . . .
. . . . . .
. . . . X . .
. . O . X X .
. . O . O X O
. X O . X O X
```

```
Player X move: (1, 5)
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . O . .
. X . . X O X
```

```
Player O move: (2, 2)
. . . . . .
. . . . . .
. . O . X . .
. . O . X X .
. . O . O X O
. X O . X O X
```

–1

3 Task 3 - 3p

Define the `query_player(game, state)` function to make the game more interactive and display the available/legal moves.

Conclusion

Submit your assignments by the deadline, file *.ipynb* where you add explanations about your implementation.