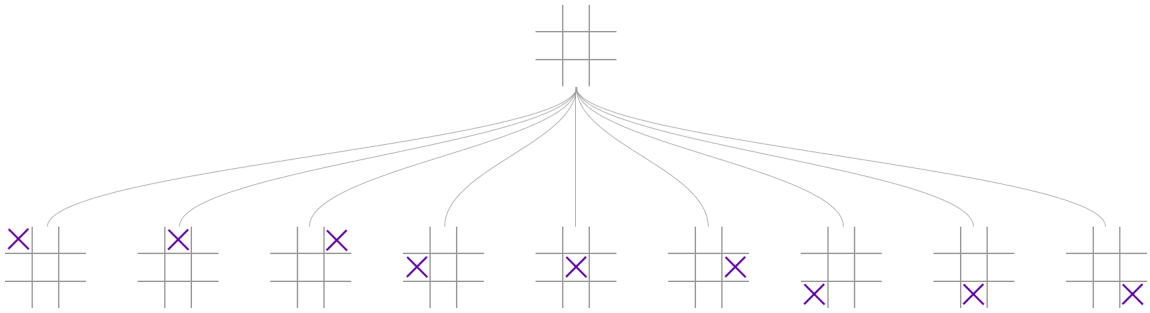
Project: Monte Carlo Tree Search Mini Problem Set

- In this project, you will learn and implement the Monte Carlo Tree Search (MCTS) algoritm on the Tic Tac Toe game.

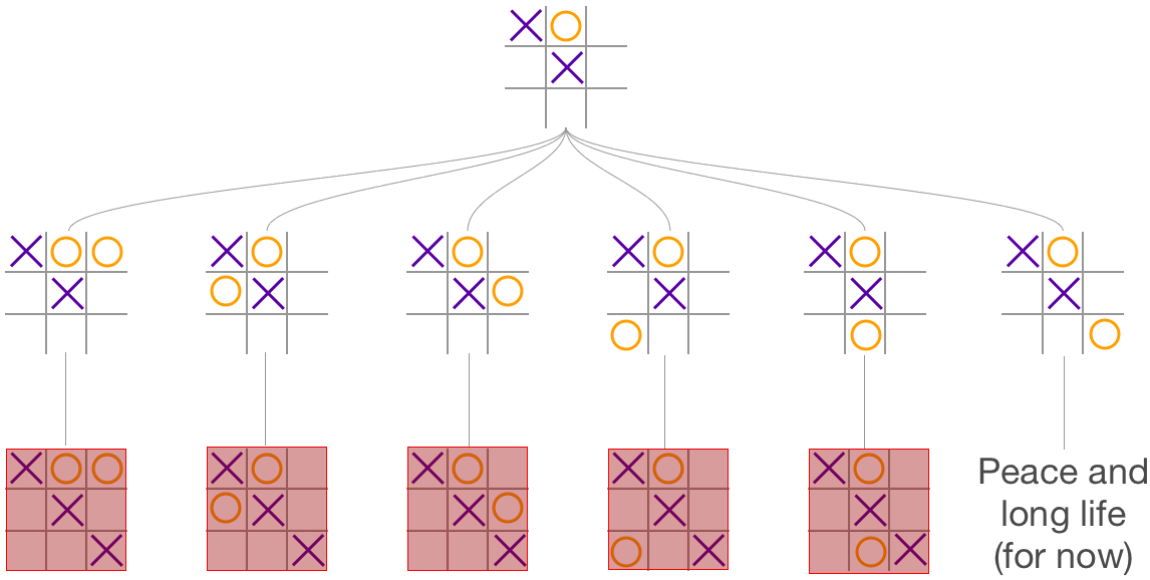
What is a tree search?

Trees are a special case of the graph problems previously seen in class. For example, consider Tic Tac Toe. From the starting blank board, each choice of where to draw an X is a possible future, followed by each choice of where to draw an O. A planner can look at this sprawl of futures and choose which action will most likely lead to victory.



Breadth First Tree Search

In a breadth first tree search the planner considers potential boards in order of depth. All boards that are one turn in the future are considered first, followed by the boards two turns away, until every potential board has been considered. The planner then chooses the best move to make based on whether the move will lead to victory or to defeat. In the following example, a breadth first search would identify that all other moves lead to a loss and instead pick the rightmost move.



Monte Carlo Tree Search

The problem with breadth first search is that it isn't at all clever. Tic Tac Toe is one of the simpler games in existence, but there are nearly three-hundred sixty thousand possible sets of moves for a BFS-based planner to consider. In a game with less constrained movement, like chess, this number exceeds the number of atoms in the known universe after looking only a couple of turns into the future. A Breadth First Search is too tied up with being logical and provably correct. Monte Carlo Tree Search leaps ahead to impulsively go where no search has gone before. In simpler terms, BFS is Spock while MCTS is Kirk.

MCTS performs its search by repeatedly imagining play-throughs of the game or scenario, traveling down the entire branch of the game tree until it terminates. Based on how this play-through went, MCTS then updates the value of each node (move) involved based on whether it won or lost the playthrough. The Monte Carlo component comes from the fact that it chooses moves at random, not based on heuristics or visit count. This nondeterminism greatly increases the potential space it can explore even though its exploration will be much less rigorous.

For example, let's look at the BFS tree above. The bad red moves have a high probability of loss because in 1/5 of the playthroughs X will instantly win. The correct, rightmost move will have a lower probability of a loss because X doesn't have this 1/5 chance of winning. MCTS will discard the red moves because of this higher loss probability, assigning them lower values whenever it selects them during a randomized play-through.

MCTS Algorithm

Check out the paper.

Problem Set API

Before you write your very own MCTS bot, you need to get sped up on the API you will be using. But even before getting into the API, please run the following to import the API and some boilerplate code:

In [ ]:
```
import algo
import sim
import random
import time
from tests import *
from game import *
```

The `Board` Class

The `Board` class is a template class that represents the state of a board at a single point in a game. We've created a `ConnectFourBoard` subclass that handles the mechanics for you. For any `Board` instance `board`, you have access to the following methods:

`board.get_legal_actions()`: Returns a python set of `Action` class instances. Each element in the set is a valid action that can be applied to the `board` to create a new `Board` instance. See the `Action` API section below.

`board.is_terminal()`: Returns `True` if `board` is an endgame board. Returns `False` otherwise.

`board.current_player_id()`: Returns an integer that represents which player is expected to play next. For example, if this method returns 0, then the player who is the first player in some simulation of a game should be the next one to play an action. This is used internally in the `Simulation` class for bookkeeping, but you will need it when you do the backup step of the MCTS algorithm.

`board.reward_vector()`: Returns a n-element tuple, where n is the number of players, that contains the rewards earned by each player at this particular `board`. For Connect Four, $n = 2$. Thus this method may return something like `(1,-1)`, meaning the player with ID 0 had a reward of 1, and the player with ID 1 has reward -1.

The `Action` Class

The `Action` class is for representing a single action that is meant to alter a `board`. We have written a `ConnectFourAction` subclass for you. Instances are hashable, so to check if two actions are the same, `hash(action1) == hash(action2)` can be used. For any `Action` instance `action`, you will only need the following method:

`action.apply(board)`: Given a `Board` instance `board`, returns a new `Board` instance that represents the board after that action has been performed. If the `action` cannot be applied, an error is thrown.

The `Node` Class

The `Node` class represents a single node in the MCTS tree that is constructed during each iteration of the algorithm. You will be interacting with this class the most. If you remember the algorithm, each node contains certain pieces of information that's associated with it. For any `Node` instance `node`, you have the following methods at you disposal:

`Node(board, action, parent)`: The constructor takes three arguments. First, a `Board` instance `board` that the node will represent. Second, an `Action` instance 'action' that represents the incoming action that created `board`. Finally, a `Node` instance `parent`. For a root node, you would pass `None` in for both `action` and `parent`.

`node.get_board()`: Returns the `Board` instance that `node` is representing.

`node.get_action()`: Returns the incoming `Action` instance.

`node.get_parent()`: Returns the parent `Node` instance.

`node.get_children()`: Returns a list of `Node` instances that represent the children that have been expanded thus far.

`node.add_child(child)`: Add a `Node` instance `child` to the list of expanded children under `node`.

`node.get_num_visits()`: Returns the number of times `node` has been visited.

`node.get_player_id()`: This just returns `board.current_player_id()`, where `board` is the board that was passed into the constructor.

`node.q_value()`: Returns the total reward that the `node` has accumulated. This reward is contained in a variable `node.q` that you can access if it needs to be changed during the algorithm.

`node.visit()`: Doesn't return anything, but increments the internal counter that keeps track of how many times the `node` has been visited.

`node.is_fully_expanded()`: Return `True` is all children that can be reached from this node have been expanded. Returns `False` otherwise.

`node.value(c)`: Returns the calculated UCT value for this node. The parameter `c` is the *exploration* constant.

The `Player` Class

The `Player` class represents, you guessed it, a player. You won't have to actually deal with this class at all in this problem set. It exists for running the simulation at the end. However, if you interested, you may look at `game.py` to see what methods are used.

The `Simulation` Class

The `Simulation` class is used for setting up a simulation for multiple players to play a game. You again don't need to worry about this class, as it is for running the simulation at the end. Refer to `game.py` if your curious about how it works.

Problem Set Code

In the following parts, we will ask you to implement the Monte Carlo Tree Search algorithm to beat the bot. You will be implementing the pseudocode starting on page 10 of the MCTS paper

Default Policy

The first step is to implement the default policy, which plays through an entire game session. It chooses actions at random, applying them to the board until the game is over. It then returns the reward vector of the finished board.

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

Browne, et al.

Note: You can use `random.choice(my_list)` to select a random item from `my_list`

In [ ]:
```
##########################################################################
# randomly picking moves to reach the end game
# Input: BOARD, the board that we want to start randomly picking moves
# Output: the reward vector when the game terminates
##########################################################################
def default_policy(board):
    # TODO
    while board.is_terminal() == False:#NOT A endgame
        board = random.choice(list(board.get_legal_actions())).apply(board)
    return board.reward_vector()
```

In [ ]:
```
test_default_policy(default_policy)
```

```
test passed
test passed
test passed
```

Tests passed!!

Tree Policy

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

Tree Policy Pseudocode (Browne, et al.)

The tree policy performs a depth-first search of the tree using `best_child` and `expand`. If it encounters an unexpanded node it will return the expanded child of that node. Otherwise, it continues its search in the best child of the current node.

The `best_child` function find the best child node if a node is fully expanded. It also takes the exploitation constant as an argument.

The `expand` function expands a node that has unexpanded children. It must get all current children of the node and all possible children of the node then add one of the possible children to the node. It should then return this newly added child.

Best Child

function BESTCHILD(v, c)

return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$

Browne, et al.

Note: For convenience, we've implemented a function that returns the heuristic inside the max operator. Look at the function `node.value(c)` for the `NODE` class API and save yourself the headache.

In [ ]:
```
############################################################
# get the best child from this node (using heuristic)
# Input: NODE,  the node we want to find the best child of
#        C,     the exploitation constant
# Output: the child node
############################################################
def best_child(node, c):
    childs = node.get_children()
    value = -9999
    ret_child = None
    for child in childs:
        if child.value(c) > value:
            value = child.value(c)
            ret_child = child
    return ret_child
```

In [ ]:
```
test_best_child(best_child)
```

```
test passed
```

Tests passed!!

Expand

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'

Browne, et al.

In [ ]:
```
############################################################
# expand a node since it is not fully expanded
# Input: NODE, a node that want to be expanded
# Output: the child node
############################################################
def expand(node):
    # TODO
    board = node.get_board()
    newaction = random.choice(list(board.get_legal_actions()))
    subboard = newaction.apply(board)
    subchild = Node(subboard, newaction, node)
```

```
    node.add_child(subchild)
    return subchild
```

In [ ]:
```
test_expand(expand)
```

Tests passed!!

```
test passed
```

Tree Policy

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

Browne, et al.

In [ ]:
```
############################################################
# heuristically search to the leaf level
# Input: NODE, a node that want to search down
#        C, the exploitation value
# Output: the leaf node that we expand till
############################################################
def tree_policy(node, c):
    # TODO
    while (node.get_board().is_terminal() == False):
        if (node.is_fully_expanded() == False):
            return expand(node)
        else:
            node = best_child(node,c)
    return node
```

In [ ]:
```
test_tree_policy(tree_policy, expand, best_child)
```

```
test passed
```

Tests passed!!

Backup

Now its time to make a way to turn the reward from `default_policy` into the information that `tree_policy` needs. `backup` should take the terminal state and reward from `default_policy` and proceed up the tree, updating the nodes on its path based on the reward.

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

Browne, et al.

In [ ]:
```
###################################################################
# reward update for the tree after one simulation
# Input: NODE, the node that we want to backup from
#        REWARD_VECTOR, the reward vector of this exploration
# Output: nothing
###################################################################
def backup(node, reward_vector):
    # TODO
    while (node != None):
        node.visit()
        node.q = node.q_value() + reward_vector[node.get_player_id()-1]
        node = node.get_parent()
```

In [ ]:
```
test_backup(backup)
```

```
test passed
```

Tests passed!!

Search (UCT)

Time to put everything together! Keep running `tree_policy`, `default_policy`, and `backup` until you run out of time! Finally, return the best child's associated action.

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($v_0, 0$))

Browne, et al.

In [ ]:
```
###################################################################################
# monte carlo tree search algorithm using UCT heuristic
# Input: BOARD, the current game board
#        TIME_LIMIT, the time limit of the calculation in second
# Output: class Action represents the best action to take
###################################################################################
def uct(board, time_limit):
    start_time = time.time()
    root = Node(board, None, None)
    while (time.time() - start_time) < time_limit:
        # TODO
```

```
        node = tree_policy(root,3)
        newreward = default_policy(node.get_board())
        backup(node,newreward)
    return best_child(root,0).get_action()
```

In [ ]:
```
test_uct(uct) # this test takes 15-30 seconds
```

```
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=1,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=3,row=3)
ConnectFourAction(color=B,col=4,row=1)
ConnectFourAction(color=B,col=2,row=2)
ConnectFourAction(color=B,col=2,row=2)
ConnectFourAction(color=B,col=2,row=2)
ConnectFourAction(color=B,col=2,row=2)
ConnectFourAction(color=B,col=2,row=2)
ConnectFourAction(color=B,col=2,row=2)
ConnectFourAction(color=B,col=2,row=2)
ConnectFourAction(color=B,col=2,row=2)
```

Tests passed!!

The Final Challenge

Time to show Stonn the power of human ingenuity! Win at least 9 out of 10 games to triumph!

In [ ]:
```
sim.run_final_test(uct)
```

Player 1 won

Player 1 won

Player 1 won

Player 1 won

Player 1 won

Player 1 won

Player 1 won

Player 1 won


```
Player 0  won
```


```
Player 1  won
```

You win!!

Stonn sits back in shock, displaying far more emotion than any Vulcan should.

"Cadet, it looks like your thousands of years in the mud while we Vulcans explored the cosmos were not in vain. Congratulations."

The class breaks into applause! Whoops and cheers ring through the air as Captain James T. Kirk walks into the classroom to personally award you with the Kobayashi Maru Award For Excellence In Tactics.

The unwelcome interruption of your blaring alarm clock brings you back to reality, where in the year 2200 Earth's Daylight Savings Time was finally abolished by the United Federation of Planets.