CSCI 0410/1411 Spring 2025
Final Project Part 2: Anytime Algorithms

Anytime Algorithms

You will continue your implementation from part 1, no new stencil code is required. You will be implementing two new `Agents`: `IterativeDeepeningAgent` and `MCTSAgent`. Both agents will be implemented in `agents.py`, and we have provided you with stencil code to get you started. You may make any other changes you see fit, other than changing the agent method headers.

In this part of the final project, you will implement two *anytime* algorithms. An *anytime* algorithm is an algorithm that can be interrupted at any point during its run and still return an answer. The longer the algorithm is allowed to run, the better the answer returned will be. The agents you implemented in Part 1 had to obey the time constraints provided, but they did so by limiting the depth of their search to some constant value to always return an answer within the cutoff time. In this phase of the final project, your agents will be able to use the `time_limit` in a more intelligent way.

This part of the final project only consists of implementing 2 agents and running an experiment to create a figure, however MCTS can be challenging to implement and debug. The stochasticity of MCTS often means that results of the search can change each time you hit run and errors won't show up in the same place each time. Give yourself enough time to work through the assignment and **understand the code that you're writing**.

Part 2A: Iterative Deepening Search

Algorithm 1 Iterative Deepening Search for Adversarial Games

```
1:  function IterativeDeepeningSearch(position, max_time)                ▷
2:      bestMove ← null
3:      depth ← 1
4:      while time remaining do
5:          value, move ← AlphaBeta(position, depth, −∞, +∞, time_remaining)
6:          if move ≠ NULL then
7:              bestMove ← move
8:          end if
9:          depth ← depth + 1
10:     end while
11:     return bestMove
12: end function
```

Task 2A: Implement `IterativeDeepeningAgent` in `agents.py` using $\alpha\beta$-pruning (or `Minimax`, if your $\alpha\beta$-pruning has errors) as its core search algorithm. You will need to implement a new variation on $\alpha\beta$-pruning that **does not take more than the amount of time remaining**. You will be evaluated based on the `IterativeDeepeningAgent` and its `get_move` function, but can add any additional helper functions you

desire. You can test your agents with the game runner file, by running the game runner file and specifying the command line arguments `agent1-type` or `agent2-type` to be ids.

For example:

```
python game_runner.py --agent1-type ids --agent2-type greedy --mode tournament --num-games 4
```

A collection of helpful tips and notes:

- You will have to build in checks to $\alpha\beta$-pruning to check if the time limit is approaching and interrupt the method before it runs out of time (i.e., stop searching further).
- Your agent must return a move *before* the maximum allotted time elapses. If the agent is given 1 second to make a move and returns a move after 1.0001 seconds, the agent will lose automatically. Build in a cushion for your agent.
- If IDS runs out of time part way through searching at depth $d + 1$, it should return the best action seen at depth d.
- The time passed to the `get_move` method of an agent is the **maximum** amount of time the agent can take. The game begins with 15 seconds available to each player and agents gain an additional second for every move made. You do not have to use all 15 seconds on move one. While testing, you may find it more enjoyable to use a small amount of time for each move and get results faster (i.e., always run IDS for 0.1 seconds).
- Game runner prints many stats after running a round of games. They were overkill for Part 1, and you likely ignored everything other than the end score. Many of these stats are related to *timing*, which will be more useful in this assignment.

Part 2B: Monte Carlo Tree Search

The performance of heuristic adversarial search algorithms on Go is relatively limited. They can beat other simple agents, but would struggle to beat a competent human. Why is this? First, Go has a large branching factor (even on this small 5x5 version of Go), limiting the search depth to only a few moves. Second, it is difficult to design a good heuristic function for Go. Material count (i.e., number of stones) is not a good indicator of performance, and most of the time territory is not settled until the end game, which means that territory early in the game is not very informative.

To address these two challenges, most Go engines use Monte-Carlo-Tree-Search (MCTS). MCTS does not rely on a heuristic function. Rather, it implicitly builds its own heuristic function by collecting statistics over many many **rollouts**, i.e., Monte-Carlo simulations of complete game play.

The goal of a search algorithm in a game is to find a good move, given a game state. Unlike the adversarial search algorithms we studied at the start of the course, MCTS does not search the game tree directly. Rather, it builds an auxiliary search tree rooted at the current game state, annotating nodes with informative statistics (e.g., the number of wins and losses). It then proceeds to simulate promising paths through this search tree until it dead ends (i.e., reaches a leaf node), at which point it expands the tree to incorporate as-yet-unexplored parts of the game. When a desired time has elapsed, MCTS returns an action at the current state that depends on the nodes' annotations.

More specifically, while time remains, MCTS runs four methods: 1. it selects a leaf node in its tree to expand; 2. it adds the children of that leaf node to its tree; 3. it simulates a game through each child; and 4. it backpropagates the results of those simulations to nodes higher in the tree. Further details of each step are presented below, and pseudocode can be found in Appendix A.

Selection

During the selection step, MCTS finds a node in its search tree to expand. This step begins at the root of the MCTS search tree (the current state of the game), and selects moves according to a **tree policy** until a leaf node is reached. Once a leaf is reached, it is returned to be used in the next step (Expansion).

But what might make for a sensible tree policy? The more simulations that MCTS runs from a state, the more information it gathers about the quality of that state. Ideally, MCTS would like to run more simulations on "good" states and not waste time evaluating "bad" states. At the same time, MCTS has to balance running simulations for "good" states with exploring as-of-yet unexplored parts of the game tree. You don't want to miss finding an even better state, just because you focused on a single good state too much. One simple way for MCTS to trade off between exploration and exploitation is to use an ε-greedy tree policy.

If ε-greedy is used as the tree policy, MCTS would proceed at a (non-leaf) state as follows:

1. With probability $1 - \varepsilon$, select an action that leads to a state with the highest win-rate among possible next states; with probability ε, select another action randomly.
2. Transition to a new node in the MCTS search tree according to the rules of the game.
3. If the new node is a leaf node, terminate. Otherwise, goto to step 1.

There are many other possible tree policies. The one most commonly used in practice is called **UCT**. UCT is an application of **Upper Confidence Bound (UCB)** applied to trees. UCB is a method for balancing the exploration-exploitation trade off. It offers both theoretical guarantees and good performance in practice. UCT computes a value for each child node s_i as follows:

$$\mathrm{UCT}(s_i) = \frac{w_i}{n_i} + c\sqrt{\frac{\ln(N)}{n_i}}$$

where w_i is the total number of wins for rollouts that went through node s_i, n_i are the total number of rollouts that went through node s_i, c is a constant that balances exploration and exploitation, and N is the total number of rollouts that went through the parent of s_i.

Notice, that the first term of $\mathrm{UCT}(s_i)$, is a value estimate of of that state. If the win-rate is 1, then the node is a very high quality state. If the win-rate is 0, then it is a low quality state. The second term is the exploration term and biases the selection towards states that have not been visited frequently compared with their siblings. The parameter c is chosen to balance these exploration and exploitation terms. The most common setting for c is $\sqrt{2}$, but fine tuning this parameter might lead to better performance.

A simple tree policy is then to simply compute the UCT values of all children of the current node and transition to the node with the highest UCT score.

Expansion

When MCTS encounters a leaf node in its search tree, it adds the children of that leaf node to its search tree. Some implementations of MCTS add only a single node (chosen at random) to the search tree, but we recommend inserting *all* possible children (all legal actions).[1]

Simulation

Simulate a rollout corresponding to the action taken by every child added during expansion, and record the results. **Note:** The states encountered during these rollouts are *not* added to the MCTS search tree. The algorithm is concerned only with the *results* of the rollouts.

[1] When only one child is added during the expansion step, the definition of a leaf node is no longer a node with no children, but rather a node what does not have a child for every available action.

In "pure" MCTS, the rollout policy is simply uniform random. However, more useful simulation results may be realizable if a better rollout policy is used (e.g., `GreedyAgent` that captures if a capture is available). Here, you will implement "pure" MCTS. In your final submission for Part 3, however, you can try to use better rollout policies.

Backpropagation

MCTS uses the results of its simulations to update (backpropagate) the values of the nodes along the simulation path further up in its search tree. Beginning with the newly expanded nodes and iterating up from there, each node in the search tree is updated with the result of that simulation (i.e., who won). Each node in the search tree stores both the number of wins for the player whose move it is and the total number of times that node was visited (the total number of rollout games that traversed that node).

Terminating

MCTS is an anytime algorithm, like Iterative Deepening. This means that we can stop at anytime and return the action of the best child node of the root. The longer MCTS runs, the more simulations are run, and the lower the uncertainty for each action. After a certain amount of time has passed or a maximum number of simulations has been run, MCTS can terminate and return the best action found.

What is the best action? Is it the action with the highest win probability? Take two nodes in the search tree with wins/total of 2/3 and 66/100, which action is better? 2/3 has a higher win percentage $66.6\% > 66\%$, but many fewer total simulations. It's possible that if you were to run more simulations on that node, its value would change significantly (there's high uncertainty). Therefore, the node with 66/100 is typically the better option as there is lower uncertainty in the value of that node.

The most common strategy for MCTS is to return the node with the highest number of total rollouts. So does that mean we don't care about win percentage at all? Because of the way MCTS selects nodes, using a tree policy that balances exploration and exploitation, it tends to run more simulations for nodes with higher win percentages. Selecting an action with the most rollouts is a means of reducing the variance of the results for the returned action.

Part 2b Tasks

1. Implement MCTS by implementing the `get_move` function in `MCTSAgent` in `agents.py`. Doing so will require that you implement the four functions that MCTS comprises. We have provided a `MCTSNode` that you can use to build a tree.

2. Use `game_runner.py` to compare the performance of your `MCTSAgent` and your adversarial search agents. For example, the following command will run a 10 game tournament between an `MCTSAgent` and an `IterativeDeepeningAgent`.

   ```
   python game_runner.py --agent1-type mcts --agent2-type ids
       --mode tournament --num-games 10
   ```

   In half the games, `agent1` will be the first player to move; in the other half, `agent2` will be the first player to move. Summarize your results in 1–2 sentences in your README.

3. Although MCTS may be (somewhat) straightforward to implement, the behavior of MCTS agents can be difficult to decipher. Your final task in this part of the project is to create a table or figure (e.g., using `matplotlib`) that help explain how your MCTS agent makes decisions. Explain your findings in a short paragraph in your README. Some experiments you may find useful to investigate: 1. How does changing c change the distribution of # visits? 2. For a given state, create a heatmap of # rollouts or win-rate to show which actions are being simulated the most. 3. How does the strength of MCTS change as the cutoff-time is changed? How different are the win rates for each node after

0.1 second of searching vs. after 1 second of searching? You do not need to run any of these, but the figure you produce should help us (and you) understand how MCTS is making decisions.

1 Submission

Submit your assignment via Gradescope.
To submit through GitHub, follow this sequence of commands:

```
git add -A
git commit -m "commit message"
git push
```

Now, you are ready to upload your repo to Gradescope.
Tip: If you are having difficulties submitting through GitHub, you may submit by zipping up your hw folder.

1.1 Rubric and Grading

Task	Points	Details
Iterative Deepening Search	25	Points awarded for correct implementation of IDS. `get_move` should *always* return an action within the time limit (so long as some small amount of time has passed to finish search with depth=1).
MCTS	45	Points awarded for `MCTSAgent`'s performance against Greedy, Alpha-Beta, and staff implementation of `MCTSAgent`. Performance should be within acceptable threshold (i.e., win almost every game against Greedy, most games against Alpha-Beta, and be roughly equivalent to staff MCTS.
MCTS Experiment/Figure	20	Points awarded for informative and clear figure. The figure should be explained clearly in the README and staff should understand what the figure shows by looking at the figure and reading the description.
README Questions	10	Points awarded for including necessary answers and descriptions in README.

A MCTS Pseudo Code

Algorithm 2 MCTS (state, π, ρ)

Input: a state in the game tree, a tree policy π, and a rollout policy ρ
Output: an action

```
node ← MCTSNode (state)        ▷ Creates a node in the MCTS tree, given a state in the game tree
while time-remaining do
    leaf ← SELECT (node, π)
    children ← EXPAND (leaf)
    result ← SIMULATE (children, ρ)
    BACKPROPAGATE (results, children)
end while
return The action corresponding to a child of state with the most visits in the tree rooted at node
```

Algorithm 3 SELECT (node, π)

Input: a node in the MCTS search tree and a tree policy π
Output: a leaf in the MCTS search tree

```
while !isLeaf (currNode) do
    if currNode is Terminal then
        return currNode
    end if
    currNode ← π (currNode)
end while
return currNode
```

Algorithm 4 EXPAND (leaf)

Input: a leaf node in the MCTS tree
Output: the children of the input leaf node

```
if leaf is terminal state then
    return [leaf] ▷ If a terminal state is in our search tree, we still need to backpropagate that information
end if
children ← [ ]
state ← leaf.state
actions ← state.legalActions()
for action in actions do
    children.append(transition(state, action))
end for
return children
```

Algorithm 5 SIMULATE (children, ρ)

Input: children: a list of MCTS nodes and a rollout policy ρ
Output: A simulation result for each child node in children

```
results ← [ ]
for child in children do
    result = run the rollout policy ρ starting at child until terminal state is reached
    results.append (result)
end for
return results
```

Algorithm 6 BACKPROPAGATE (results, children)

Input: A new leaf node (children) and simulation result for each new leaf node (results)

```
for child in children, result in results do
    currNode ← child
    while currNode ! = NULL do
        currNode.visits ← currNode.visits +1
        if result == WHITE–WIN  &  currNode.player == BLACK then
            currNode.value ← currNode.value +1
        else if result == BLACK–WIN  &  currNode.player == WHITE then
            currNode.value ← currNode.value +1
        end if
        currNode ← currNode.parent
    end while
end for
```

Note for Backprop: Values should be high when the currNode was a good action to take from the parent node. Therefore, when the result is WHITE-WIN and the current player is BLACK (the parent node was WHITE), we increment value.