CSCI 0410/1411 Fall 2024
Final Project: Go Engine

Milestones	Release Date	Due Date	Due Time
Part 1 (Search)	11/15	11/25	5:59 pm ET
Part 2 (Learning)	12/2	12/9	5:59 pm ET
Final Bot & Writeup		12/17	11:59 pm ET
Tournament Ends		12/19	11:59 pm ET

1 Introduction

Go (*a.k.a.* Wieqi, Igo, or Baduk) is a two-player zero-sum game of strategy popular throughout the world. Game-playing agents have long been a focus of the AI community. Indeed, the term "machine learning" was dubbed by Arthur Samuel in 1959, while building a Checkers-playing agent that learned by playing against itself. Since then, AI researchers have capitalized on Samuel's idea of self-play to build successful game-playing agents for many parlor games, including Backgammon (TDGammon, 1992) and Chess (DeepBlue, 1997). Nonetheless, until very recently, it was thought that Go-playing agents that could perform as well as humans were out of reach. When Alpha-Go defeated the world champion Lee Sedol in 2016, it shocked the AI and Go communities alike. In this project, you will implement game-playing agents for the game of Go. You, too, may be shocked by how well your agent learns to play!

2 Learning Objectives

The goal of the final project is to help you synthesize the material you've learned throughout this course. We will provide structure for Parts 1 and 2, but the techniques you choose to use for Part 3, your final agent, will be up to you.

What you will be able to do:

- Combine techniques from different subfields of AI to produce a strong Go-playing agent
- Design experiments to evaluate the performance of different game-playing agents, and summarize the results in easy-to-interpret tables or plots

What you will understand:

- How Monte Carlo Tree Search uses randomization to discover a good policy
- The tradeoffs between heuristic adversarial search and Monte Carlo Tree Search

3 Setup

This project requires that you install Open-Spiel, a library developed and maintained by Google Deepmind for research on learning and games. It contains implementations of many popular games, including Go. You can install it with the following command:

```
pip install open_spiel
```

4 Go

Like Checkers and Chess, Go is a two-player, zero-sum, deterministic, alternating-move game. Also like Checkers and Chess, Go is played on grid. Two players alternate placing colored stones on the grid's intersection points, called **territory**. The first player to move places a black stone, while the second player places a white stone. The two then alternate until the game ends. If one player completely encircles any of the other's stones, the encircled stones are captured, and removed from the board. At any point, a player may pass. The game ends when no legal moves are available, or if both players pass consecutively. When the game ends, the players tally their total territory.[1] The player who has surrounded more territory wins.

Go can be played on a variety of board sizes, ranging from as small as 5x5 (beginner play) up to 19x19 (professional play). In this project, your agents will play on 5x5 and 9x9 boards.[2]

The player who moves first (Black, in Go) typically has a significant advantage, as in Tic-Tac-Toe, Connect 4, Chess, etc. To account for this advantage, additional points, called komi, are added to White's score at the end of the game. We will be using a komi of 5.5 in this project for 9x9 games. This means that Black has to score 6 or more points better than White to win. For games on a 5x5 board, a komi of 0.5 is used. If a fractional value is used for komi, the game cannot end in a draw.

We highly encourage you to try out an interactive Go tutorial, such as this one, to better understand the rules of Go. There are also many video introductions to Go available online.

4.1 Time Controls

Many competitive board games are played with per-move or per-game time limits; without such limits, games could take forever! These time limits are often referred to as *time controls*. At the start of each player's turn, a clock starts counting down. After the player makes a move, the opponent's clock begins. If a player's remaining time runs out before a move is made, that player forfeits the game.

Our games will use **incremental** time control, in which each player is allocated an initial block of time (in this project, 15 seconds), plus an additional increment of time after every move (in this project, 1 second). For example, if a game lasts 200 moves, then the total number of seconds of search time would be 215 seconds. Time management, meaning how a player uses the available time, is key to a successful Go agent.[3]

5 Part 1a (Warm Up): Heuristic Tree Search

Your first task in this project is to create a set of simple benchmarking agents to have on hand for evaluating the more sophisticated agents you will build later.

Our implementation of Go implements the same abstract `AdversarialSearchProblem` interface we used in Homework 2, when you built minimax and $\alpha\beta$-pruning agents to play Tic-Tac-Toe and Connect 4. As a result, you can import your implementations of minimax and $\alpha\beta$-pruning into your final project code base.

As the state space in Go is vast (e.g., there are 81 intersections on a 9×9 board, making for 2^{81} states), it will not be feasible to run minimax and $\alpha\beta$-pruning without limiting the depth of the search. Recall that the depth-limited versions of these algorithms require a heuristic, i.e., a means of evaluating the quality of an arbitrary state. We have provided a very simple heuristic, `GoProblemSimpleHeuristic`, to jump start your use of your existing adversarial search implementations. This heuristic evaluates a state by computing the difference between the number of black and white stones on the board. We also include an implementation of `GreedyAgent`, which uses `GoProblemSimpleHeuristic` to greedily select an action.

In Homework 2, your agents could take as long as you were willing to wait for them to make decisions. In contrast, in this project, time is not unlimited. Since you are allocated an additional one second (beyond the initial 15) for every move, the easiest way to make sure your agent does not run out of time is to make

[1] There are many other variations of scoring for Go, such as counting captured pieces as part of the score.

[2] If you want to play against your agent, the GUI is much faster on 5x5 boards. RL is also much faster on 5x5 boards.

[3] and to a successful Brown student!

sure it uses less than one second per move. For minimax and $\alpha\beta$-pruning, one way to do this is to find a search depth that always terminates within one second.

However, fixing a search depth throughout a game is very limiting. At the start of the game, Black, the first player, has 81 legal moves.[4] The second player then has 80 legal moves, which yields a total of $(81)(80) = 6840$ possible states after the first two moves. Later in the game, when there are, for example, 20 available moves, searching to a depth of 3 plies would search the same number of states as just 2 plies at the start of the game ($(20)(19)(18) = 6840$).[5]

Rather than fixing a search depth in advance, an alternative is to build an agent that searches for a fixed amount of time. Assuming there is just enough time to search 6480 states, such an agent would lookahead through depth 2 at the start of Go, and through depth 3 after 20 moves. This approach can be realized using an **anytime** algorithm, which, as the name suggests, can return a solution at any time! When the time comes, an anytime algorithm simply returns the best solution found so far.

Earlier, you were tasked with discovering a search *depth* for all moves, now you are tasked with discovering a search *time* for all moves. The simplest approach would be to set the time limit to 15 seconds for the first move, and 1 second thereafter, as an additional second is added to the players' clocks with each move. But you can do better! For example, if you somehow knew that a game would take k moves, then you could spend $15+k/k$ seconds per move.

And there are better time management strategies still, the most sophisticated of which might determine a time limit based on the game state, e.g., how many moves have been played so far, how well the agent perceives its positioning on the board, etc. You will have the opportunity to implement more sophisticated time management strategies in later parts of this project.

For now, you will be porting your implementations of minimax and $\alpha\beta$-pruning to your Go code base, so that you can include them in your initial experimentation. You will find, however, that their performance is limited. You will therefore be implementing an anytime algorithm called Iterative Deepening Search (IDS), which iteratively runs a "core" search algorithm (e.g., $\alpha\beta$-pruning) with an increasing cutoff depth until a solution is found or time runs out. We provided you with an implementation of IDS in Homework 1. You will have to modify it slightly to work with $\alpha\beta$-pruning and a cutoff time.

Part 1a Tasks

1. Run 100 games of `GreedyAgent` vs `RandomAgent` using the command:

   ```
   python game_runner.py --agent1-type greedy --agent2-type random
       --mode tournament --num-games 100
   ```

   A tournament outputs the following statistics:

   - Score for each agent, i.e., total wins − total losses
   - Score as Black (first mover) for each agent
   - Statistics about the amount of time used by each agent to find its moves

   What are the results? Do the scores surprise you at all? Report your findings in your README.

2. Play a few games against the `GreedyAgent` using the command:

   ```
   python gamerunner.py --agent1-type greedy --mode gui
   ```

   Note: Use the arrow keys to select a cell to play and use the enter key to take an action.

   What is `GreedyAgent`'s strategy? Why, across multiple games, is it always more or less the same? What is the problem? Describe your findings in your README.

[4] Ignoring *pass* for simplicity.
[5] Assuming no captures took place and no player passed.

3. Modify `GreedyAgent` to randomly select among all actions with the best heuristic value. Does this alter its performance against `RandomAgent`? Why are the results so different? Explain your results in your README.

4. Implement `MinimaxAgent` and `AlphaBetaAgent` in `agents.py`. As these agents implement the `GameAgent` interface, all that is required is that you implement the `get_move` method for each agent. This method should simply call your implementations of `MiniMax` and `AlphaBetaSearch` from Homework 2.

    - We have included command line arguments for `AlphaBetaAgent` and `MCTSAgent` in `game_runner.py`. If you'd like to use `game_runner.py` for new agents, you can add them to the `create_agent` method of `game_runner.py`. Add `MinimaxAgent` to `create_agent` if you'd like to use it with `game_runner.py`.
    - Tip: In this project, the terminal values of the game are either $+1$ or -1. In contrast, in Connect 4 in Homework 2, the values were $+\infty$ or $-\infty$. You may have to adapt your Homework 2 implementations slightly to handle this change. We recommend converting the terminal values of $+1$ and -1 to $+\infty$ or $-\infty$ within your implementation of $\alpha\beta$-pruning.

5. Experiment with both of your agents to discover a search depth that obeys Go's time constraints. What is this search depth for each agent? Include your answers in your README.

6. Implement `IterativeDeepeningAgent` in `agents.py` using $\alpha\beta$-pruning as its core search algorithm. Recall that IDS searches at increasing depths until the cutoff time is reached, at which point it returns the best action seen. If IDS runs out of time part way through searching at depth $d+1$, it should return the best action seen at depth d.

6 Part 1b: 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 9x9 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 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).

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.

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). In Part 1 of the project, you will implement "pure" MCTS. In your final submission, however, you can (and should) 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 1b 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.

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` that searches to depth 3.

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

   In half the games, `agent1` will be the first player to move; in the other half, `agent2` will be the first player to move. You can adjust the number of games played in the command if you wish. Summarize your observations in 1–2 sentences in your README.

3. Although MCTS is 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 (or multiple tables or figures) that help explain how your MCTS agent makes decisions. Explain your findings in a short paragraph in your README.

7 Downloads

You can access the support and stencil code for this assignment through Github Classroom.

7.1 Support Code

- This project uses `GameAgent`, `GameState`, `AdversarialSearchProblem`, and `GameUI` from Homework 2. These classes can be found in `adversarial_search_problem.py`
- `GoSearchProblem.py`: Provides implementations of `GoState` and `GoSearchProblem`, which implement `GameState` and `AdversarialSearchProblem`, respectively.
  - One thing to note is a (perhaps unintuitive at first) design choice made by the developers of OpenSpiel, namely that `Actions` in `GoSearchProblem.py` are integers, not pairs of (x, y) coordinates. In particular, the actions available on an empty 9×9 Go board are $\{0, 1, \ldots, 81\}$. The first 81 of these actions (0 through 80) actions correspond to locations on the board and the final action (81) corresponds to pass. To convert from an integer action a to coordinates on an $n \times n$ board, you can compute $(x, y) = (a \mod n, \lfloor a/n \rfloor)$.
- `heuristic_go_problems.py`: Contains an implementation of `GoProblem` and a *very* simple heuristic `GoProblemSimpleHeuristic`, which you can pair with your `MinimaxAgent` and `AlphaBetaAgent`.
- `game_runner.py`: Contains methods for running games between two agents, and tournaments among multiple agents.
- `go_gui.py`: Provides an implementation of `GameUI`, for visualization and user interaction.
- `go_utils.py`: Contains utility functions for setting up adversarial search problems.

7.2 Stencil Code

- `agents.py`: Where you will implement your agents. `GameAgent` defines an interface that your agents must implement. The core of the agent behavior is the `get_move(state, time_limit)` method, which returns an `Action` given the current game state and the remaining time on the clock.

8 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.

8.1 Rubric and Grading

Task	Points	Details
Heuristic Search Agents	10	Points awarded for implementing the `get_move` function in `MinimaxAgent` and `AlphaBeta` agent
Iterative Deepening	15	Points awarded for correct implementation of Iterative Deepening. Must find a solution within the cutoff time.
MCTS Agent	45	Points awarded for correct implementation of MCTS. Implementation must include the four functions MCTS comprises. Your `MCTSAgent`'s performance should be similar to ours.
MCTS Figure	20	Your figure and explanation will be graded based on two criteria: 1) is it easy to interpret, and 2) is it informative. In sum, your figure should help you understand `MCTSAgent`'s behavior.
README Questions	10	Points awarded for answering each README question

A MCTS Pseudo Code

Algorithm 1 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 2 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
    currNode ← π (node)
end while
return node
```

Algorithm 3 EXPAND (leaf)

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

```
children ← [ ]
state ← leaf.state
actions ← state.legalActions()
for action in actions do
    children.append (transition(state, action))
end for
return children
```

Algorithm 4 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
    results.append (result)
end for
return results
```

Algorithm 5 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 == BLACK–WIN  &  currNode.player == BLACK then
            currNode.value ← currNode.value +1
        else if result == WHITE–WIN  &  currNode.player == WHITE then
            currNode.value ← currNode.value +1
        end if
        currNode ← currNode.parent
    end while
end for
```