CSCI 0410/1411 Fall 2024
Final Project Part 3: Go To Town with your Go Agent!

So far, we have provided guidance on the design and development of your agents. Your agents are competent, but can be significantly improved. Your final task is open-ended and your goal is to create your own agent to participate in a class-wide tournament with TA bots and your peers' agents. You have complete freedom within the guidelines provided. You can use MCTS or not. You can use deep learning or reinforcement learning to learn policies or construct your own hand-crafted heuristic function. The choice is yours. We provide some places to start and other potentially useful resources, but these are not required of your agent.

1 Partners

You can work with one partner for this final submission if you wish. You will submit through only one person's Gradescope. If you work with a partner, the intention is that you can experiment and try more things than working alone. We therefore have higher standards for the report and agent.

2 Submitting your Agent

To construct your final agent, we will call `get_final_agent_5x5()` or `get_final_agent_9x9()`. You must add these methods to your `agents.py` file and they should construct your final agent when called (or raise NotImplementedError). We provide the example methods here:

```
def get_final_agent_5x5():
    """Called to construct agent for final submission for 5x5 board"""
    raise NotImplementedError

def get_final_agent_9x9():
    """Called to construct agent for final submission for 9x9 board"""
    return GreedyAgent() # construct agent and return it...
```

We will be running two concurrent sections of the tournament for 5x5 games of Go and 9x9 games of Go. If your approach works on both board sizes, you should submit both agents. Learning-based approaches will likely only work for one size. If your approach is specific to one board size, you need only submit one final agent and leave the other method not implemented.

You will submit your agent and all necessary files to Gradescope just like any other assignment. Gradescope will evaluate your agent against a GreedyAgent with a simple heuristic, just to test that your agent does not time out or otherwise have an error. If your agent successfully defeats the GreedyAgent, then it will automatically be added to the tournament.

Each day, starting at midnight, a daily tournament will be held among all agents submitted and the results will be reported on our website. The name of your agent (set by implementing the `__str__` method of your agent) and a unique ID from gradescope will be used to report results.

3 Rules

- Your agent **must** make moves within the specified time limit. If you are running a loop with `while curr_time - start_time < time_limit`, your agent will always run out of time! If you run out of time for a single move, your agent will forfeit the game and it is an automatic loss. We encourage you to give a bit of lee-way just to ensure your agent to not lose on time.
- No *pondering*: Pondering refers to the practice of thinking during your opponents turn. While generally interesting to think about how you might do this, we want to avoid any nefarious and sabotaging behavior (spooling up empty work during your opponents turn to starve them of computational resources). Your agent will persist between moves (i.e., you can save information between moves), but it should not continue processing after `get_move` ends.
- No catching of `support.TimeoutException` (or generic exceptions). If your agent takes longer than the time cutoff, we will terminate it and your agent will forfeit the game. If your agent attempts to evade this termination (by catching the exception we will interrupt with), you will be disqualified from the entire tournament.
- Must be written in Python. Search algorithms work better the more states they can explore. Python is a relatively slow language compared to compiled languages, like C and C++. Yes, your agents would be more efficient if written in a faster language, but that is not in the spirit of the class. The focus should be on AI techniques used, not programming language used.
- Your agents will run in docker containers with the same setup as the local environment that you've installed (if you followed the directions). This means that if you install additional libraries locally, they may not be available remotely for the tournament. If your agent works on Gradescope, it should work in the tournament as well.
- Your submission to Gradescope should not take more than 50 MB of memory. This means if you use additional files for neural network parameters or an opening book, they should fall within this 50 MB limit. We don't expect people to come close to this limit.

4 Possible Extensions

- Better Heuristic for Iterative Deepening Search: What is currently missing from our non-learned heuristic values are an idea of which player has captured the most territory. The simple heuristic is only tracking the number of pieces captured. You can produce a better heuristic by finding regions of empty cells that are completely surrounded by a single color (or the edge of the board).
- Opening Book: In one sense, the first move of the game is the hardest move for your search algorithms. The number of available actions and therefore the branching factor of the search tree is the highest on the first move. In another sense, the first move should be the easiest move of the game. Your agent should not need to use any computation time on the first move of the game, you should simply hard code in the best action to take on the first move. But why stop there? There are known popular good openings for both 5x5 and 9x9 Go. These can be memorized and added in what is called an *Opening Book*. For an example of such openings, see this website. To implement this, your agent will need to check if a state is in your opening book and find the best action (also in the opening book) if it is. Consider what data structures are required for fast look ups of a given state. Your agent can load other files into memory, we just ask that you limit your total memory usage to a reasonable amount (see the rules).
- Transposition Tables: A transposition in Go (and other board games) is a state that can be reached through multiple move orders. For example, the shown game state shown in Figure 1 may have been reached in multiple ways. Black could have played their stone at (6, 2) first and the stone at (8, 3)

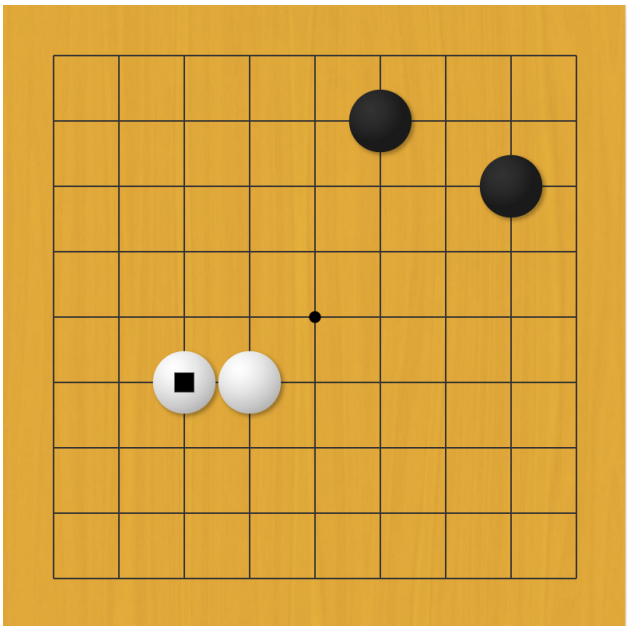

Figure 1: A board state that can be reached in multiple ways.

second, or vice versa. In your existing search algorithms different move orders result in different states. However, you can reduce the branching factor of your search tree significantly if you merge together search nodes that correspond to the same game state. A Transposition Table saves states and their current evaluations so you can check to see if a state has already been evaluated and you can avoid any expensive re-computations.

- Saving Information Between Moves: the `get_move` of your agent will always be called after a single move by your opponent. You may be able to reuse parts of your old search tree or other data structures between different moves. Since your agent is an object, you can use instance variables (`self.whatever`) to save information in between calls to `get_move`

- Flexible Time Management: When there is obviously only one good move (e.g., you can immediately capture a large number of pieces), your agent shouldn't waste time building a large search tree. Most of the time spent thinking by top-level professional Go and Chess players is in the middle game. Perhaps these professionals have the right idea. You may want "flexible" time management that doesn't take a constant amount of time for every move. Remember, the `time_limit` passed to `get_move` is not how much time your agent *has* to take, it is how much time it *can* take.

- Better selection/playout policies in MCTS: As discussed previously, playouts in MCTS may be more informative if a simple heuristic is used to guide actions rather than uniform random simulations. Likewise, other selection methods (or parameters) may be used to improve MCTS performance as well.

- More Supervised Learning: We provided you with an existing dataset for 5x5 games that we collected using MCTS. You can further improve these methods by using better neural network architectures or by gathering more data to train on. To collect more data, you can run your own agents against each other and collect information on states, actions taken, and the results of the games.

- Reinforcement Learning: Similar to supervised learning, the goal will be to learn a heuristic function or policy for a given state. The advantage of reinforcement learning is that you can potentially collect more data (through many simulations) than you can with supervised learning alone. Open-spiel provides a

gym-like environment (much like the environments we used for blackjack and cartpole), available by calling:

```
from open_spiel.python.rl_environment import Environment
env = Environment("go", board_size=5)
```

If you wish to try Deep-Q learning, there are many tricks that are often used to improve the stability of the training procedure. Even though neural networks are more expressive than the linear models we used in our previous fitted-q learning assignment, they can be much harder to train. The library Stable Baselines provides a helpful list of tips and tricks to get started on selecting an RL algorithm for your problem.

AlphaGo and AlphaZero both used self-play to train their agents. In self-play, your reinforcement learning agent would play against itself, slowly improving over time (hopefully). We recommend you start by playing your reinforcement learning agent against a simple foe, like RandomAgent or GreedyAgent, to begin with and incrementally increasing the difficulty as training progresses (an approach called curriculum learning).

- Hybrid Agent: Each of the agents you developed in parts 1 and 2 of the final project have their strengths and weaknesses. Maybe your learned agents from part 2 play the opening and middle phase of the game well, but fail towards the end of the game (which is what ours tend to do). Alpha-beta search is guaranteed to find the best moves available, but in general runs slowly when there are many available actions. You can combine the strengths of these two agents by using a learned policy in the beginning of the game and switching to alpha-beta search with a non-learned heuristic at the end of the game.

5 Additional Resources

- The ChessProgrammingWiki is a community focused on the development of Chess engines. Many of the techniques used in chess engines are applicable to Go as well. Additionally, they have a page dedicated to Go. For instance, it has pages dedicated to opening books and transposition tables that you may find helpful if you choose to implement these features.
- A thesis by Petr Baudis on using MCTS to play Go. Includes practical advice for handling transpositions, selection policies, and playout policies. https://pasky.or.cz/go/prace.pdf
- Alpha Go: provides a description of the methods used to achieve super human Go play for the first time.

6 Tips and Hints

- "Premature Optimization is the Root of All Evil" – Donald Knuth. Faster implementations will explore more states and generally perform better. However, beware of trying to optimize your code too soon. It is better to have a bug-free slow implementation than a fast buggy implementation. If you cannot understand your own code, you will never be able to get it to run without bugs.
- Version Control (git) is Your Friend: You may find yourself implementing many different versions of your agent and comparing their performance. You will help yourself if you use helpful commit messages and commit whenever you have a notable agent worth saving. You will thank yourself if you ever have to revert to a previous version of your code.

- Running experiments and creating figures is not just helpful for understanding results, it can help you debug your code faster. It can be hard to debug the algorithms you are developing with break points or print statements alone (as you are probably aware). **Making a figure, plot, or other visual representation can help you debug your implementation** and understand whether or not your algorithm is working as intended.

7 Grading

We are running a tournament where you can track your performance against student and staff agents. However, we value creativity and ambition above pure performance and will take a holistic approach to grading. We encourage you to try something that may not work (for example, reinforcement learning) over something that is guaranteed to work (e.g., MCTS with better time management and other incremental improvements). You will be evaluated along the following four axes:

- Approach and Creativity: Is your approach incremental or did you try something new and experimental?
- README: Does your README clearly explain your approach and how you evaluated your approach? Does it contain any helpful figures to help you understand how your algorithm is working?
- Code Quality: Is your code well written and easy to follow? Is it your own original code?
- Performance: Does your agent perform well against other agents.

You need not check off each of these items to achieve a good score on the final project (the README is always required). If your README is thorough and helps explain your unique approach, you need not have an agent with high performance. If your agent performs well and is unique, we will forgive your poorly written code.

7.1 Final Report

You **must** include a README with your final submission that describes your agent and what you have done in this final phase of the final project. If you have known bugs, you should include a description of them here. If you tried other approaches that are not a part of your final submission, you should include them here and a brief statement on why you decided not to use them (e.g., they were too slow, buggy, etc.). If you work with a partner, you must briefly describe what each person contributed to the project.