CSCI 0410/1411 Fall 2025
Final Project Part 3: Go To Town with your Go Agent!

Due Date: December 14th, 11:59pm

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 Tournament Brackets (i.e., AI usage)

We will be running **two separate** tournament brackets for the final tournament: one where AI use is not allowed and one where you can use any amount of any AI you'd like. The AI-enhanced bracket will be run on `9x9` boards and the "human" bracket will run on `5x5` boards. The time controls will be the same for both sections, namely 15 seconds at the start with a 1 second increment per move. The other difference between the two brackets will be on how they are graded. Human submissions are graded on 4 criteria, including creativity, code quality, readme explanations, and performance. The AI bracket will be graded almost entirely on performance, how well it does against your peers and staff bots. You should read this entire handout before deciding which bracket to enter. When you submit your code, you will submit either a `9x9` agent or a `5x5` agent. Because of computational limitations when running the tournament, we will only let students submit an agent to one tournament (you can't have an agent in both tournaments, even though it may be interesting to do so).

Very Important: Since we are explicitly providing an option for AI usage, there will be severe consequences for students who enter the human bracket with code developed by an AI or a README file written with AI. Please don't create extra work for the course staff and deans, if you are going to use AI, don't submit to the human bracket.

2 Submitting your Agent

To construct your final agent, we will call `get_final_agent_5x5()` (or `get_final_agent_9x9()` for AI bots on a 9x9 board), which should return an agent *object*. In the Stencil, this method currently returns an `MCTSAgent`, but you can and should change this to whatever your final Agent is. **These methods will be how we know which bracket you are submitting to.** If you submit a `get_final_agent_9x9()` that returns an agent, we will assume you are submitting to the `9x9` tournament. If you submit a `get_final_agent_5x5()` that returns an agent, we will assume you are submitting to the `5x5` tournament. If you submit a file with both implemented, we will assume you are submitting to the `9x9` tournament.

You can name your agent by implementing the `__str__` method, although agents will also be identifiable by your gradescope ID. The easiest way to ensure your agent behaves as expected is to put all required methods and classes directly in the `agents.py` file. Do not modify `GoSearchProblem` or any other files, as

we will provide staff implementations of these to both agents playing a match. (Our implementation does randomize legal actions, as was suggested in part 1 of the project)

You will submit your agent to Gradescope just like any other assignment. Upon submission, your agent will be uploaded to a repository with the other agents, and will play in the next iteration of the tournament. You will be given a unique ID by the autograder tied to your agent.

Each day, starting at midnight, a daily tournament will be held among all agents submitted and the results will be reported on our website. This tournament will use your latest submission (i.e., you can only have one agent in the tournament at a time and it will be taken from your most recent gradescope submission). 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.

After the final deadline, we will start a larger (i.e., more games played) final tournament that will run for a few days. It is challenging to add agents to the tournament while the tournament is running. Therefore, we will not be accepting **any** late submissions for this final deadline. Any submissions that are late will not be graded on performance and will receive a 0 in that category.

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 agent will be provided with 8 CPU cores and 16 GB of RAM to use. We will not provide GPUs to agents to use. If you'd like to use deep learning methods, you can still train your models on GPUs (through colab or other means), but at execution time we will run every agent on the same hardware configuration.
- 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 100 MB of memory. This means if you use additional files for neural network parameters or an opening book, they should fall within this 100 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. These can be memorized and added in what is called an *Opening Book*. 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).

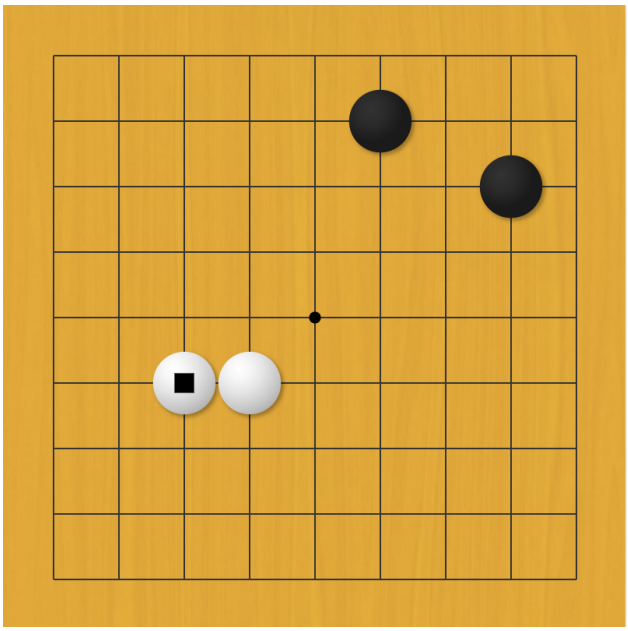

Figure 1: A board state that can be reached in multiple ways.

- 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) 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. Humans don't play `5x5` go that often, but they do play `9x9` Go, so datasets can be found online for the larger tournament size. You can further improve these methods from part 1 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 would 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)
  ```

  We do not have an equivalent for PyGo, however openspiel is available for installation on Google Colab or Github codespaces as well.

  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. You may be able to combine IDS and MCTS in similar ways.

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

- The AI-enhanced bracket will use `9x9` boards by default. If you are going to enter that bracket, you can run games on a `9x9` board by sending the `--size` argument to game runner. Or, you can save time and potential mistakes by directly editing the default game size in `game_runner.py` (i.e., change `parser.add_argument('--size', type=int, default=5, help='Board size')` to have a default size of 9.
- "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. Now is the time to learn how use git to help development, not just for how to make final submissions.
- 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.
- The daily tournaments will provide feedback on how well your agent is performing, but it will be quite infrequent and should not be the only measure you use to determine how well your agent works. So how should you evaluate how well your agents are performing? When you've produced an agent you're happy with, save a copy of that agent in `agents.py`, then compare it with new agents you develop. You started with a `RandomAgent`, then compared a `GreedyAgent` to that first agent to verify that it worked. Then you developed a minimax agent and compared with the `GreedyAgent` to verify it worked. Then you developed more agents and compared to the other previous agents. Your development will look a bit like a ladder, with every agent being compared to previous agents.

7 Grading

Note: These grading guidelines base some amount of the grade on performance against your peers. However, we cannot predict the size of these tournament brackets ahead of time, so we will adjust the grading scales up if we think it is needed because too few people submitted to one of the brackets.

7.1 Grading for human tournament bracket

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? Is it clear you put thought and effort into your agent?
- 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 well documented? Is it your own original code?
- Performance: Does your agent perform well against other agents (staff and peers)?

You need not check off each of these items to achieve a good score on the final project (the README is always required). For instance, 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.

Each of the categories described above will contribute equally to your final score. Your final score will be based on a total of 100, even though 120 points are possible. This means you can lose points in some categories and still achieve a good overall score.

Task	Points	Details
Approach and Creativity	30	Full points awarded for attempting something novel and ambitious (i.e., not just small things from the list of suggestions provided). Reusing agents from previous parts of the final project without **any** modifications will not receive any points in this category.
README	30	Full points awarded for a thorough explanation (including experiments and figures) of how your agent behaves. Submissions with no experiments or figures can still achieve high marks in this category, but the full 30 points will be reserved for exceptional READMEs.
Code Quality	30	This category will also rely, in part, on the description of your approach in your README. We will look at your code and make sure you are following good style and that the code was written by you, or has citations to outside sources where appropriate. We expect most people will get full marks in this category.
Performance	30	Students who rank in the top half of student submissions will receive 30 points in this category. Remaining students with working agents will receive points based off of a linear scale from 30 to 10 based on the ranking of their agent. So the submission that ranks at the 25'th percentile (halfway between the last ranked submission and the cutoff for full points) will be awarded 20 points. Students who do not submit a working agent (i.e., there are runtime exceptions or syntax errors) will receive a for this category.

7.1.1 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.).

7.2 Grading for AI-enhanced Tournament

In the AI-enhanced tournament, we care much less about creativity and code quality. The only metrics we have to measure your project is the quality of the agent and your report on the agent. In the AI-enhanced bracket your grade will be out of the 100 available points.

Task	Points	Details
README	20	Full points awarded for a thorough explanation of how your agent behaves. Your README should describe how you interacted with your AI tools (e.g., which ones you used and some prompt examples) as well as how your agent actually works and any relevant references.
Performance	80	Top 10% of submissions: 100% credit Top 25%: 95% credit Top 50%: 85% credit Top 75%: 75% credit Bottom 25%: 65% credit

7.2.1 Final Report

You **must** include a README with your final submission that describes your agent and how AI was used to create it. You should include things like: "this is the prompt I used to get it started..." and how you improved the agent over time by prompting AI to improve it or having it produce suggestions for directions for your own implementation to go in.