Introduction

Such a complex project of designing - and implementing - a Python bot to play the game Diplomacy has taught me extensive new ideas about artificially intelligent agents, in addition to a deeper understanding of Python as a programming language. Countless hours of designing, writing, debugging and rewriting code has culminated in a project that, while not entirely *good* at what it does, contains many meticulously tuned interacting components, representing a significant growth in my development as a programmer. The lofty goal of winning 100% of games that I set myself at the start of this project has sadly remained wholly unrealised, yet I emerge immensely proud of what I have accomplished. Please enjoy a laboured, and oftentimes heavy-handed, account of my struggles, insights and ultimate successes, with statistics of specific functions featured in the file test_24214277.py.

Diplomacy

Upon opening the project document, I was overcome by an unmistakable sense of dread, as I realised that in addition to creating an immensely complex and intricate agent, I would also have to learn an entirely new strategic board game. Not one to back down from a challenge, I headed straight to the Web Diplomacy simulation to begin to understand the rules and found initial satisfaction in moving units in tandem and forming simple plans.

I was unfortunately destined to be swiftly executed by Italy.

Several rather frustrating games followed, during which I gained a deeper understanding of the more complex rules involving convoys and supports, and an even deeper understanding of the inevitability of utterly dismal performance.

Tired of being defeated after less than half a decade, I turned to the Python library implementation, although it did little to ease my despair. Confusingly specific order notations in string format, combined with a dauntingly dense and menacing set of sample agents, I spent a solid week simply understanding the game engine, its available methods and return formats, ever-so-slowly putting together an understanding of the game on which 40% of my unit depended upon.

First Ventures

Inspired by the greedy agent, my initial foray into agent-controlled Diplomacy took the form of a **simple greedy agent** concerned only with moving across the board to pick up vacant supply centres. Disregard for coordination, support or even allied positions, this bot performed remarkably well against static agents - if your definition of *well* is avoiding a loss in an unlosable game. These remarkable results took the form of Russia, Austria, England and France each almost doubling their centre count (average of 6.2), while unfortunately Italy and Turkey fared much worse as they grappled with their inherent inability to dislodge other units. Inspired by this early promising result, I moved to develop a more rigorous **order set generator.**

And thus began the arduous process of getting bots to move with any semblance of sensibility.

To avoid the need to wait for the extremely slow test.py file to run dozens of tests, I created my own “shortTest.py” file that would simply run a single game against a pool of agents of my choosing. To simplify my bot creation, I restricted all opponent agents to be instances of the StaticAgent provided in the agents_baselines.py file, so that I would have perfect information about their future moves. A fully observable, deterministic and episodic environment for my agent provided a good way for me to test my agent’s ability without needing to predict enemy moves, but unfortunately even this simple environment proved a challenge.

Improvements

Extending upon a simple exploratory agent, I looked to develop a simple **greedy move evaluation function**. My theory was that if the agent could rank all available moves (as provided by the Game instance) and choose the one with the highest evaluation, a dramatic increase in skill would result. Hours upon hours of wrangling with dictionaries and list comprehensions and navigating function return types and weighting supports and calculating enemy threats and counting supply centres later, I had a simple move evaluation function that returned an integer valued score for any given action. Truly, I have gained a new appreciation for documentation and type hinting - there are only so many times one can print a dictionary of lists of lists to figure out my indexing before their sanity begins to degrade, although the PrettyPrint module is genuinely a lifesaver.

As such, I was deeply dismayed to see that the performance of the bot had barely improved (5.8 to 7.1 supply centres across all powers).

As hindsight is so often twenty-twenty, I realised that of course the agent could not perform much better than simply moving randomly - any attempt to dislodge an enemy unit from a supply centre (ranked extremely highly by the move

evaluation) was entirely unsuccessful, as there was no corresponding support order. Each agent calculated its evaluation in isolation, entirely locally, and as a result no overall process was made.

Multiple Order Sets

In an attempt to improve the synergy of unit behaviour, I experimented with **generating multiple complete sets** of orders and evaluating them in terms of how successful they would be if they were to run. To calculate this hypothetical evaluation, I performed a **singly-ply search** where each order set would be simulated, a fast evaluation of the resulting position was carried out, and the greatest evaluation was the order set that was chosen.

This fundamental design would remain throughout the evolution of this project. Conceptually, I believe that it is empirically sound - generate some candidate order sets, evaluate them, and pick the best candidate. However, one challenge remained prescient - how can good candidate order sets be picked?

Combinatorics

It was at this point where I encountered the first major hurdle - combinatoric explosion. Naively, I believed that I could simply generate all possible combinations of actions for my units, simulate performing each of them with predicted enemy moves (strictly holds for now), and return the best order set. Indeed, at the start of the game, this method was working wonderfully. My first test was picking up supply centres with ease, and all was going swimmingly until the second winter phase, where the bot was able to build another five units - and the program hung.

With three units, the number of possible move combinations is 3x3x3 = 81, and as such the program had ~~no~~ minimal trouble simulating these, but as soon as eight units were present, a total of 8x8x8x8x8x8x8x8 = 16,777,216 order sets were being evaluated. Yikes.

I realised that I was going to have to prune some order sets. Even with only a single prediction of enemy moves, and a single depth of search, I had vastly too many moves to check - the branching factor of my game tree was exponentially large.

Interaction Graph

After some thought (and research on similar but strictly different multi-agent games), I had the idea of creating an **“interaction graph”** of units - units able to influence each other, either through support or attacking, would generate an edge in a graph, where nodes represent the units themselves. As such, taking the connected components of this graph would allow me to narrow down the number of order combinations to smaller groups, and combine them after. However, despite success in countries such as France where I observed units splitting into distinct groups, more central countries such as Italy remained as a single connected component, doing nothing to reduce the amount of candidate order sets. As such, I abandoned the use of the connected components to implement another promising technique I had researched - **beam search.**

Beam Search

To me, beam search is such an incredible algorithm. Similar to A*, a heuristic (in my case the sum of individual order scores plus a calculated “synergy bonus”, factoring in attacks receiving support, etc.) is used to expand upon promising partial order sets. However, where A* maintains a score for all nodes in the search, beam search is more directed, only keeping the top-K promising nodes and discarding the rest. As such, the search maintains a directed “beam”, ideally finding the best, or at least multiple promising, solutions.

It was with great difficulty and a large amount of trial and error that I was able to construct a sensible **“synergy bonus” function**, which simply works by evaluating whether or not a new action would utilise the other supports/moves within the partial order set, or cause problems such as self-bounces. By simply sorting all beams by this evaluation and maintaining only the top-K nodes (a beam width of 100-200 worked best through experimental results), the final top 20 candidate order sets were returned.

In the end, my beam search constructed prospective partial orders sets one unit at a time, and I found it beneficial to sort by units who had a single order that was much better than the rest (i.e. a unit surrounded by enemies absolutely must hold its position), and iteratively build a full order set, pruning after each unit’s actions had been added. I believe this is due to the first orders having a larger effect on the beam’s evaluation, as poor initial orders will cause the beam to be pruned too early. Early pruning is the major downside of this technique, however after many tweaks to **evaluation weightings** and **different sorting techniques** etc., the resulting order sets are very sensible.

After many tweaks and changes to sorting and scoring, the agent was generating extremely sensible and coherent order sets, and I was able to move on to increasing my search depth.

The Game Engine

Now that I had 20 prospective order sets, I created a **depth first search algorithm** to simply duplicate the Game state, perform the actions, and then continue from there, up to a depth of 2 turns ahead. At the end of this search, the greatest initial order set would be chosen and returned. But as things in this project seemed to inevitably go, I was met with a significant problem - the built in diplomacy engine is slow. Like, extremely slow.

Even simulating a single turn ahead and running my candidate move generation was taking as much time to generate order sets as it was to simply clone the game instance, and situations with five units was taking around thirty seconds per more to evaluate, rather comically eclipsing the single second time budget.

As such, I knew I had but one choice - to **rewrite the engine**. I wasn't concerned with more complex rules such as multi-fleet convoying or support-cutting, I just simply wanted a performant and lightweight game instance that I could instantiate many times without significant overhead. I am still rather unsure as to what the expected path for us to take was - was I just simply using the game copy wrong? But since I thought it was imperative that some forward planning and simulation be undertaken, my only choice was to spend a few days creating a simplistic Game class, capable of resolving simple orders and returning unit positions and available moves. To be entirely honest, this was one of the more straightforward sections of the project, and thankfully it was orders of magnitude more performant than the built in engine, albeit slightly less accurate.

Monte-Carlo Tree Search

I knew from the beginning that I may have to implement a **Monte-Carlo Tree Search (MCTS),** and after realising the speed (or lack thereof) of my candidate order generation, I felt that the only way to achieve long term planning and coherence was to simulate multiple turns into the future using much cheaper move generation methods. Initially, **random moves** provided a decent win rate of 60% and 15% in scenario 1 and scenario 2 respectively (by this point I had resigned to the fact that anything other than a 100% loss rate in scenario 2 is a major win), but I decided that a **lightweight likely-move generator** would be better. Using a simple heuristic of individual order evaluation (like before), but combining it with selecting a random option from the top candidate moves to move often generate support and attack pairs, performance rose to ~75% and ~25% win rates.

My MCTS uses **UCB-1** to explore nodes for rollouts, with an exploration constant factor of 1.4, as that provided decent performance after testing. At the end of its iteration, as timed out by the timeout decorator to ensure under 1 second of runtime, the best child is returned and used as the best order set.

Conclusion

In conclusion, my agent performs extremely poorly. Despite dozens of hours tuning evaluation functions, rewriting engine code and simulating thousands of likely positions, the bot fares on-par with simple greedy strategies, and would most certainly be improved, or at least matched, if I spent all of a turn's time budget on the beam search with a wider beam and took the top result. I had dreams of probabilistic determinations of likely enemy orders, Bayesian networks, and I spend plenty of time researching intriguing sounding topics such as Hierarchical Task Networks and Goal-Oriented Action Planning solutions. But alas, I was defeated by the extensive time I spent debugging and testing different values for evaluation functions.

I understand that the rubric specifically asks for two basic techniques and two new techniques (and a limit of 3 pages), but throughout this project I experimented with numerous different algorithms and design choices as the project evolved, indicated in **bold** throughout this report. A more detailed account of my specific evaluation function values and abandoned algorithms can be found in my experiments file, as well as a link to the GitHub repository tracking my changes over time. Due to how slow running the simulations were to get detailed information, a large amount of trial and error remains unfortunately untracked.

Although this agent doesn't perform as well as I had hoped, I would rather have a carefully thought-out and reasoned-about project containing numerous different interacting algorithms and components that wins most games, than a single-ply greedy agent that may perform slightly better with a better individual order evaluation function. I remain immensely proud of my achievement in completing this project, and sincerely hope that you have enjoyed reading this report detailing my journey.