Over the course of this semester I have worked to create a project where I can compare different AI algorithms and the solutions they create to solve a problem. That project was my own recreation of the game *Snake* in Python, where several different AI agents using algorithms I have implemented competed against each other to determine which algorithms performed better under different scenarios.

There are many different ways one can approach the problem of beating the game of Snake, such as meticulously taking the shortest path to the fruit, or carefully tracing patterns to ensure an obstacle is never encountered. I wanted to build agents that would display similar ideas in strategy, so I had to choose appropriate algorithms that could leverage speed and safety. There are three algorithms that stood out to me the most for solving this problem, and I have created an agent dedicated to each one—those algorithms are the A* (A-star) search algorithm, and the Q-Learning and Approximate Q-Learning reinforcement learning algorithms.

The A* search algorithm is a pathfinding and graph traversal algorithm, known for its efficiency and accuracy in finding the shortest path. It combines features of two other search algorithms (Dijkstra's Algorithm and Greedy Best-First Search) to ensure the solution is always found. Specifically, the algorithm makes use of a combination of a cost function that associates a cost with every move (this ensures the lowest cost solution is found), and a heuristic function that guides the algorithm in the correct path more efficiently. This algorithm works great for Snake, as it can efficiently find the best path to reach the fruit at any given point. Its main drawback is that the algorithm doesn't exactly have a way to avoid trapping the snake. However, this drawback can be slightly mitigated if the agent guides the snake to chase the snake's tail when the fruit is not directly reachable. There is a cost associated with each move, so moves that do not guide the snake towards the fruit will be less desirable. The heuristic function used to ensure efficiency is simply the Manhattan distance from the snake's head to the fruit.

Compared to the A* algorithm, the two reinforcement learning algorithms are quite different. Q-Learning (QL) operates by iteratively updating a Q-table (a table for all state and action pairs) to learn the optimal action for a given state. It uses the Bellman equation to update Q-values based on observed rewards and penalties. QL works well for smaller state and action spaces, but becomes inefficient for larger or more continuous state and action spaces. Approximate Q-Learning (AQL) is an extension of Q-Learning that can be used when the state and action space is too large to represent Q-values as a table. Rather than computing Q-values and storing them explicitly, approximations for the Q-values are made based on defined features that are meant to best represent key points of the state. These features are updated through exploration as well, allowing AQL to scale more efficiently at the cost of precision. Since the Q-values are not stored, AQL can be used in much larger state action spaces than regular QL can. I have implemented several features into the AQL agent. There is a feature for whether the game has been won, one for the Manhattan distance to the tail, another for the Manhattan distance to the fruit, whether a fruit has been eaten, how many obstacles are up ahead, and whether the snake is currently aligned row or column wise with the fruit. These are supposed to be meaningful representations of determining the value attached to a state and action. Both versions of Q-Learning and Approximate Q-Learning aim to learn optimal policies using rewards but differ in how they represent and compute Q-values. For the problem of beating Snake specifically, the state and action space can be defined as the current positions of the snake’s body on the board, the position of the fruit, and the actions that the snake can reasonably make in the current position.

Both QL and AQL require training rounds so that the agent can reasonably learn about the world it exists to interact with. I have created two different ways to train the agents—the first way is to simply let them play through a specified number of games with a chance of making random moves occasionally to encourage exploring new states. This chance of making random moves decreases as time goes on, so that in later training stages the agent is making more of the decisions. The other training metric is one that is supposed to help guarantee that the agent has explored a variety of states. The idea is that rather than just playing a bunch of different games, the snake has to keep playing games until it consumes a specified number of fruit at each position on the grid, for each possible snake size. This ensures that the snake has been able to discover ways to reach the fruits from many positions.

In order to best compare these algorithms and agents, they will be compared specifically to test how the different algorithms perform when the scale of the environment (the grid size) is changed. To do this, an experiment will be run where each agent will get to play 10 games on grids of size 3, 5, and 7, and the results will be reported. For this scenario, comparisons can be made for how the size of the grid affects the runtimes of the algorithms, how many fruits each agent was able to get on average, the average number of moves made, and for the reinforcement learning algorithms specifically—how the number of training rounds affects performance.

My expectations for the results are as follows: I believe that the A* algorithm will perform better on larger grid sizes, and will perform especially well when the snake is smaller. I think this agent will get a decent number of fruits, but will be punished at larger sizes for going only after the fruit and not looking out for traps. I think that the two reinforcement algorithms will hopefully perform better at larger snake sizes, as hopefully the agents can learn to avoid traps. I believe these algorithms will both take longer to run however, as they have to undergo training in order to make informed decisions. Now whether AQL will outperform QL depends on how well the features are for depicting the state. If I have good features it will perform better, but otherwise QL may be better. Overall, I think that A* will be the fastest, but won't eat as much fruit as the other two algorithms—and I think that AQL will outperform QL in speed and the number of fruits obtained if the features are good.

In terms of time and space complexity, the runtime of A* depends on the heuristic, so it can be hard to approximate exactly—but for each individual run of the algorithm, it can only search as far as the bounds of the grid, meaning if it were to expand every node, it runs proportional to the size of the grid squared O(size^2) (assuming size itself is just the length or width). But, it also uses a priority queue at each iteration, which has a runtime of O(logN) for the methods, where N is no larger than size^2. So the worst case runtime is somewhere around O((size^2)log(size^2)). For space complexity, the algorithm uses a set which can store every position on the grid, as well as the space needed for the priority queue. Since the priority queue stores the value pushed and its priority, it uses extra space. The queue can store as many positions as there are in the grid, and each item pushed is a tuple of which there is a list of the path built thus far. If the entire grid is traversed in a path, then the total amount of storage used is around O(size^2).

For the reinforcement learning algorithms, their runtimes are really hard to judge, as the bulk of their runtime lies in the training, and if the agent is moving randomly it can run for a very long time. In the worst case (and unlikely) scenario the agent moves repeatedly between the same spaces forever, that's an unbounded runtime. Needless to say, both algorithms run longer than A*, and the runtime for AQL specifically is affected by what kind of features it is using to approximate Q values. For the space complexity of QL, the space is proportional to the size of the Q table, which stores the values for each possible state action pair encountered. This makes the space complexity exponentially increase each time the snake increases in size. Approximate Q learning eliminates the need for the Q table, and instead keeps track of a fixed amount of features. The largest use of storage by this agent is a set that is used to keep track of if the agent is running in a circle, which is proportional to the size of the full grid.

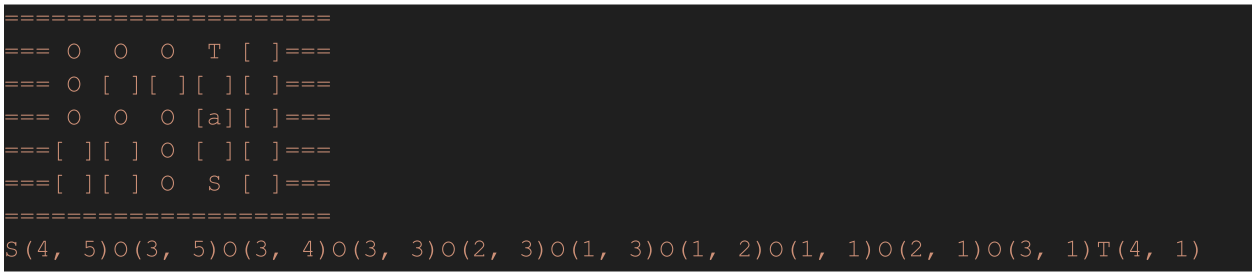
After running the actual experiment, I have found that for a 3x3 grid, the A* agent makes an average of 20 moves per game, eats an average of 7.7 fruits, and won the game 7 times. The QL agent makes an average of 16.5 moves per game, eats an average of 8 fruit, and wins 10 times. The AQL agent makes an average of 22.5 moves per game, eats an average of 7.2 fruit, and also wins 7 times. As we scale the grid size up to a 5x5 and beyond, no agents were able to win but, A* moves an average of 58.9 times and eats 13.4 fruits. QL moved an average of 72.8 times and ate 9.9 fruits. AQL moved an average of 123 times and ate 11.2 fruits per game. For the final 7x7 grid, the A* agent moved an average of 137 times and ate an average of 23 fruits. The QL agent moved 144.4 times, and ate an average of 7.3 fruits. The AQL agent astonishingly moved an average of 1259.6 times and ate 6.1 fruits per game. Due to time constraints (largely thanks to AQL taking a long time to run) I did not run the experiment on different training sizes for the reinforcement agents—they both were trained on 10000 practice games and additionally were trained to find food on each grid position 10 times for each size.

My predictions weren't quite on the mark for this experiment. I was mostly right about the performance of A*, but the reinforcement learning agents did not quite perform as well as I would have hoped (definitely my fault as the one implementing the algorithms). The QL agent didn't perform too badly, but it definitely didn't get as many fruits as A*. And interestingly, the AQL did well on the 5x5 grid but not so great on the 7x7. It's interesting how both reinforcement learning algorithms performed better on the 5x5 than the 7x7—I guess they needed more training since it was a larger space, and the features for AQL could probably be improved in general since it really did not seem to be wanting to go after the food. Overall, the A* algorithm performed the best and the fastest on average.

Ultimately, I didn't get to accomplish everything I wanted to with this project in time, but I intend to continue working on it after this report. My biggest struggle was definitely with getting approximate Q learning to work. I tried many different features, but a lot of them seemed to just make the agent avoid the fruit and not want to increase in size (such as tracking how much open space was reachable from a given position). I was hesitant to try using negative weights, but I think that could maybe have worked for some situations? I would like to study approximate Q learning much more in depth, and understand better how one chooses good features and how to scale them properly. Once I can get the reinforcement learning agents to work a bit better, I would like to create an actual GUI for this as well. This was a great learning experience, and it definitely piqued my curiosity. I would really like to practice with reinforcement learning more in other settings.

https://github.com/INFO-450-550-Artificial-Intelligence/project-fall-2024-Jgarcia713

I have my link to my repository above. The file to run is runGame.py. Currently, the program uses the console to display the state of the game (sample shown below) where S represents the head of the snake, T represents the tail, O represents the body, [a] represents a fruit, [ ] represents an empty spot, and '===' represents a wall. There is output at the bottom as well for the coordinates of each body part in the order they are connected to the snake. Currently the program is set to run the same experiment I did for the results reported earlier, and the results are written to a text file named results.txt



```
=====================
=== O  O  O  T [ ]===
=== O [ ][ ][ ][ ]===
=== O  O  O [a][ ]===
===[ ][ ] O [ ][ ]===
===[ ][ ] O  S [ ]===
=====================
S(4, 5)O(3, 5)O(3, 4)O(3, 3)O(2, 3)O(1, 3)O(1, 2)O(1, 1)O(2, 1)O(3, 1)T(4, 1)
```