**Project 1: Search in Pacman**

Parker Gray

CSC 665

**Overview**

This project’s goal is to create algorithms that help a Pacman agent find paths through the Pacman maze world to the goal state. The project is versatile in its requirements- we are both required to implement a number of different algorithms *and* solve for different goal states.

**Problem Statements**

The problem statement for *Questions 1 through 4* is fairly simple- we are required to create a *general* function that can search a Pacman maze graph (coordinate plane) for a goal state, and then utilize this general function in the various tree searching algorithms (DFS, BFS, UCS, A\*).

The problem statements for *Questions 5 through 7* is slightly more complex- Question 5 requires the implementation of a class CornersProblem that represents a Pacman search problem in which the goal state is visiting all four corners of a Pacman maze. Question 6 requires the creation of a *heuristic* (in layman’s terms, a *good estimation* of a path’s ‘cost’ without necessarily being the cost of the path) for the CornersProblem in order to help the accuracy of finding the optimal path for CornersProblem. Question 7 requires the completion of another heuristic- called foodHeuristic- for the FoodSearchProblem (that is, a search problem in which the goal state is Pacman’s consumption of all food on the screen).

**Solution Design**

For *Questions 1 through 4*, designing a solution was on the simpler side, thanks to all of the hints given in class. In creating the *general* graph search function, Prof Yoon prompted us in the direction of using her Node class that was provided in the lecture notes to represent the nodes of the graph search tree. The Node class contains all necessary information for a given search node- Pacman’s state, a parent node if the node has one, and actions that can be taken from that current node. From here, we had to provide a graph search function that investigated a search tree to find a given goal state. The following had to be kept track of- the *starting* *state* (provided in any node instance), the *goal state*, the *set of visited nodes*, and *the set of nodes following a given node instance* (provided in the node class as a function), and of course the *means in which all of the nodes viewed is held* (referred to as the fringe from now onward).

Following the completion of the general graph search, we simply had to apply the general graph search to various algorithms- DFS, BFS, UCS, and A\*. Applying the graph search to these functions is easy as pie, as the only difference between these algorithms is the type of fringe used to store nodes. Designing a solution for this part of *Q1 through Q4* was as simple as referring to the lecture notes for the knowledge of which fringe type to use!

For *Question 5*, things are slightly more complex. Some parts of CornersProblem are provided for us, like the declaration function of a CornersProblem instance that contains relevant variables to the problem- the walls of the maze, the starting position of pacman, the corners to be reached to satisfy the goal state. To solve this question we had to implement the getStartState(), isGoalState(), and getSuccessors() functions. Both getStartState and isGoalState have simple solution designs- getStartState() is a simple accessor that needs to return the information relevant to solving the problem- that is, Pacman’s goal state and the corners that have been reached. isGoalState() needs to check at every point whether Pacman has visited all four corners and needs to return true if all the corners have been reached.

getSuccessors() is a slight bit more complex- Pacman’s current position needs to be acknowledged, as well as the corners that have already been visited, and the possible successor functions for the current Pacman position. Following this, all possible actions from the current position need to be iterated through and added to the possible successor function list.

For *Question 6,* the goal is to create a heuristic that works for the CornersProblem defined above. In deciding what would be a viable heuristic for the CornersProblem, a straightforward ‘distance from a corner’ from the current position seemed easiest and turned out to work perfectly fine.

For *Question 7*, a heuristic needs to be created for the FoodSearchProblem. In creating a viable heuristic for this search problem, one needs to take into account the current Pacman position as usual as well as the locations of all food on the maze. The solution needs to take into account all possible directions that can be taken to consume the food in the most optimal way. This turned out to be a little more difficult than one would think.

**Solution Implementation**

*Questions 1 through 4* all refer to the same general graph search algorithm, so I think it is best that this algorithm be discussed first. The graph search used in the Project 1 Part 2 submission that this Documentation file is submitted in as well uses the very same graphSearch() implementation as provided in class on Tuesday, 2/20- but it doesn’t hurt to go over how it works.

graphSearch(problem, fringe) has two arguments- the problem that it is being used to solve and the fringe that it is provided to hold nodes in. graphSearch() firstly gets the startState from the problem given to it using getStartState(). It then pushes a node into the fringe that has a state value of the startState() given a line above. Following this, a variable *visited* is declared as a set that will hold all visited nodes. Then, so long as the fringe is *not* empty, the fringe is *popped*, returning its value (last one placed in if the fringe is a stack, first one placed in if the fringe is a queue).

The value popped is placed into a variable *node*, and this variable node is checked for whether its state is the goal state. If it is, its path is returned. If it isn’t, check if the node popped has a state that has already been visited. If the node has been visited, add the node to the set of visisted states, and check the node’s children, if it has any. For every child found, repeat the process described above until the goal state is found.

Following the completion of the above general graphSearch(), the implementation of DFS, BFS, UCS, and A\* only differ in the fringes used. What’s important to remember is that DFS uses a stack, BFS a queue, UCS a priority queue with cost as the priority, and A\* uses a priority queue that takes into account cost as well as the heuristic value of each node. Therefore, all one has to do for these functions is type the following:

return graphSearch(problem, fringeSpecificToFunctionType).

Fortunately for us all of the fringe types described above are provided in util.py that came with the project zip file. So, the DFS through A\* section of the *Q1* through *Q4* looks like

*DFS: return graphSearch(problem, util.Stack())*

*BFS: return graphSearch(problem, util.Queue())*

*UCS: return graphSearch(problem, util.PriorityQueueWithFunction(lambda node: node.path\_cost))*

*A\*: return graphSearch(problem, util.PriorityQueueWithFunction(lambda node: node.path\_cost + heuristic(node.state, problem)))*

*Question 5* (the CornersProblem) is a behemoth- the largest section of code in the project for certain. I’m going to get right to the function explanations rather than explaining all of the little variables given in the \_\_init\_\_() function provided to us.

getStartState() returns a tuple of the starting position for Pacman as well as an empty list that is meant to represent the list of corners already visited. It’s pretty straightforward.

isGoalState() takes the current position and corners visited from state defined in the paragraph above and verifies whether the current position is itself in the list of corners provided in \_\_init\_\_() – and if the current position *is* a corner, and is simultaneously *not* in the set of visited corners, it returns the Boolean of (the list of visited corners has a length of 4)- ie, have all corners been visited. If so, return true- the corners problem is complete. Otherwise, return false.

getSuccessors() gets the current position and visited corners list from state once more while also declaring an empty list of successors. It then iterates through all possible actions for Pacman- north, south, east, and west. The change in x and y for the currently examined action are measured as dx, dy, and the next x and y position that Pacman will travel to is create by adding the original position’s x and y cords with dx and dy respectively. This set of coordinates is recorded as the nextPos. Then, if the next position doesn’t have x and y coordinates that hit a wall, we check to see if the next position is a corner, and if it is a corner, if it hasn’t been visited already. The next position is appended to the list of already visited corners if it hasn’t been visited. Following this check, a variable successor that has form successor = (state, action, cost) is created- the successor function for this case is successor = ((nextPos, alreadyVisitedCorners), action, 1). This successor is appended to the list of successors. Following this tedious mess above, we repeat this cycle for every possible action that Pacman can take at a given point as stated- north, south, east, and west. After that’s done we return the list of successors.

*Question 6* is thankfully less complicated. Like in Question 5, both the current position and visited corners list are given by state[0] and state[1]. We also add a variable heur\_sum that will be used for the final heuristic for a path. A notYetVisitedCorners list is created and all corners that haven’t been visisted are added to this list. While the length of notYetVisitedCorners isn’t 0, minimum distance to a corner is calculated using the manhattanDistance() function in util.py. This distance is added to the heur\_sum variable. The current position is set to the corner that we examined, and we remove that corner from the notYetVisitedCorners list. The next time, distance will be calculated from the corner that is our current position. heur\_sum is returned after all of the corners’ distances have been solved for.

*Question 7* makes use of the mazeDistance() function provided in searchAgents.py. Variables position and foodGrid (of class Grid type from game.py) are created, and foodGrid is turned into foodList in order to make use of the fact that lists are iterable while foodGrid is not. Variable heuristic is declared as a list []. For all positions that contain food in foodList, heuristic appends the value of mazeDistance that takes (current position, food position, problem.startingGameState)) as its arguments. After all possible distances are appended, return the maximum value of the heuristic list.

**Solution Result/Evaluation**

All autograder.py tests as well as all question specific questions listed under each question header in Project 1.pdf pass. Most of the solutions resolve very quickly with the exception of question 7’s foodHeuristic. I imagine foodHeuristic resolves exceptionally slowly as a result of the fact it has to solve so many distance calculations for every bit of food on the map many thousands of times.

**Conclusion**

I’ll admit that I don’t know all that much python, so this project was as much about learning basic syntax as it was formulating answers to questions. I feel I still am a complete novice in regards to python, but I am slowly starting to understand the language. This project was made much easier when I realized that the other classes within searchAgents.py served as decent templates for problem 5, with FoodSearchProblem in particular being super helpful in brainstorming solutions for CornersProblem. I’m not exactly looking forward to Project 2…I assume it’s only going to get harder from here, and this project was hard enough!