CSIT6000F 2019 Fall Semester Assignment #2 - The Programming Part

Date assigned: Wednesday, Oct 16.

Due time: 23:59 on Wednesday, Oct 30.

How to submit it: Submit your written answers as a pdf file on canvas.ust.hk.

Penalties on late papers: 20% off each day (anytime after the due time is considered late by one day)

Problem 7. (20%) Programming assignment (A*). This programming assignment uses Berkeley AI Project[1] Pacman game for you to practice the A* algorithm. You'll find the relevant files/programs in the archive `assign2-program.zip`, including many testing files.

In `searchAgents.py`, you'll find a fully implemented `SearchAgent`, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan are not implemented – that's your job.

First, test that the `SearchAgent` is working correctly by running:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

The command above tells the `SearchAgent` to use `tinyMazeSearch` as its search algorithm, which is implemented in `search.py`. Pacman should navigate the maze successfully.

Now it's time to write full-fledged generic search functions to help Pacman plan routes! Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) which gets to that state.

Hint: We have implemented the breadth-first search (BFS) algorithm in the `breadthFirstSearch` function for you. If you have no idea about where to start, it should be a good reference. Since the algorithms are very similar, only a few changes are needed to implement DFS based on BFS.

Here are some general notes:

1. All of your search functions need to return a list of *actions* that will lead the agent from the start to the goal. These actions all have to be legal moves (valid directions, no moving through walls).
2. Make sure to **use** the `Stack`, `Queue` and `PriorityQueue` data structures provided to you in `util.py`! These data structure implementations have particular properties which are required for compatibility with the autograder.
3. Your code should quickly find a solution for:

   ```
   python pacman.py -l tinyMaze -p SearchAgent
   python pacman.py -l mediumMaze -p SearchAgent
   ```

[1] `http://ai.berkeley.edu`

```
python pacman.py -l bigMaze -z .5 -p SearchAgent
```

The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration). Is the exploration order what you would have expected? Does Pacman actually go to all the explored squares on his way to the goal?

4. If you use a `Stack` as your data structure, the solution found by your DFS algorithm for `mediumMaze` should have a length of 130 (provided you push successors onto the fringe in the order provided by getSuccessors; you might get 246 if you push them in the reverse order). Is this a least cost solution? If not, think about what depth-first search is doing wrong.

Tasks

1. **A* Search (10 pts)**

   Implement A* graph search in the empty function `aStarSearch` in `search.py`. A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The `nullHeuristic` heuristic function in `search.py` is a trivial example.

   You can test your A* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic (implemented already as `manhattanHeuristic` in `searchAgents.py`).

   ```
   python pacman.py -l bigMaze -z .5 -p SearchAgent\
   -a fn=astar,heuristic=manhattanHeuristic
   ```

2. **Corners Problem: Formulation (5 pts)**

   The real power of A* will only be apparent with a more challenging search problem. This task is for you to formulate a new problem called the corners problem. The nest task is to design a heuristic for it.

   In *corner mazes*, there are four dots, one in each corner. Our new search problem is to find the shortest path through the maze that touches all four corners (whether the maze actually has food there or not). Note that for some mazes like `tinyCorners`, the shortest path does not always go to the closest food first! *Hint*: the shortest path through `tinyCorners` takes 28 steps.

   Implement the `CornersProblem` search problem in `searchAgents.py`. You will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached. Now, your search agent should solve:

   ```
   python pacman.py -l tinyCorners -p SearchAgent\
   -a fn=bfs,prob=CornersProblem
   ```

   ```
   python pacman.py -l mediumCorners -p SearchAgent\
   -a fn=bfs,prob=CornersProblem
   ```

Hints You need to define an abstract state representation that *does not* encode irrelevant information (like the position of ghosts, where extra food is, etc.). In particular, do not use a Pacman `GameState` as a search state. Your code will be very, very slow if you do (and also wrong). The only parts of the game state you need to reference in your implementation are the starting Pacman position and the location of the four corners.

3. **Corners Problem: A* Heuristic (5 pts)**

   Implement a non-trivial, admissible heuristic for the `CornersProblem` in `cornersHeuristic`.

   ```
   python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
   ```

   Note that `AStarCornersAgent` is a shortcut for

   ```
   -p SearchAgent\
   -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic.
   ```

   Notice that you heuristic must be admissible, i.e. the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal (and non-negative). It should not be a trivial one like returning zero everywhere.