CS440 – Searching

Purpose: In this lab, your Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. You will build general search algorithms and apply them to Pacman scenarios.

Getting Started: Download the search.zip files and complete each of the following assignments to be submitted for grading. Each should be done individually but you can consult with a classmate to discuss your strategies or if you get an error message that you do not understand.

You should be able to play a game of Pacman by running `pacman.py`. Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain.

In `searchAgents.py` you will 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 you can test `SearchAgent` on a tiny maze with a hard coded algorithm by running `pacman.py` with the command line option:
`-l tinyMaze -p SearchAgent -a fn=tinyMazeSearch`

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.

Important note: 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).

Another important note: 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.

Hint: Each algorithm is very similar. Algorithms for DFS, BFS, UCS, and A* differ only in the details of how the fringe is managed. So, concentrate on getting DFS right and the rest should be relatively straightforward. Indeed, one possible implementation requires only a single generic search method which is configured with an algorithm-specific queuing strategy. (Your implementation need not be of this form to receive full credit).

Assignment 1 – Finding a Fixed Food Dot using Depth First Search:

Implement the depth-first search (DFS) algorithm in the `depthFirstSearch` function in `search.py`. To make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states.

Criteria for Success:

Your code should quickly find a solution when running `pacman.py` for:

```
-l tinyMaze -p SearchAgent
-l mediumMaze -p SearchAgent
-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?

Hint: 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.

You probably also will want to use the autograder with the command line `-q q1` to verify everything is working.

Assignment 2 – Breadth First Search:

Implement the breadth-first search (BFS) algorithm in the `breadthFirstSearch` function in `search.py`. Again, write a graph search algorithm that avoids expanding any already visited states.

Criteria for Success:

Test your code the same way you did for depth-first search.:
`-l mediumMaze -p SearchAgent -a fn=bfs`
`-l bigMaze -p SearchAgent -a fn=bfs -z .5`

Does BFS find a least cost solution? If not, check your implementation.

Hint: If Pacman moves too slowly for you, try the option `--frameTime 0`.

Note: If you've written your search code generically, your code should work equally well for the eight-puzzle search problem without any changes. Try running `eightpuzzle.py`.

You will also want to use the autograder with `-q q2` to verify everything is working.

Assignment 3 – Varying the Cost Function:

While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses. Consider mediumDottedMaze and mediumScaryMaze.

By changing the cost function, we can encourage Pacman to find different paths. For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rational Pacman agent should adjust its behavior in response.

Implement the uniform-cost graph search algorithm in the uniformCostSearch function in `search.py`. I encourage you to look through `util.py` for some data structures that may be useful in your implementation.

Criteria for Success:

You should now observe successful behavior in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use (the agents and cost functions are written for you):

```
-l mediumMaze -p SearchAgent -a fn=ucs
-l mediumDottedMaze -p StayEastSearchAgent
-l mediumScaryMaze -p StayWestSearchAgent
```

Note: You should get very low and very high path costs for the StayEastSearchAgent and StayWestSearchAgent respectively, due to their exponential cost functions (see `searchAgents.py` for details)..

You will also want to use the autograder with `-q q3` to verify everything is working.

Assignment 4 – A*-search:

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.

Criteria for Success:

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`).:
`-l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic`

You should see that A* finds the optimal solution slightly faster than uniform cost search (about 549 vs. 620 search nodes expanded in our implementation, but ties in priority may make your numbers differ slightly). What happens on `openMaze` for the various search strategies?

You will also want to use the autograder with `-q q4` to verify everything is working.

Assignment 5 – Finding All the Corners:

The real power of A* will only be apparent with a more challenging search problem. Now, it's time to formulate a new problem and 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.

Note: Make sure to complete Assignment 2 before working on this, because this question builds upon that work.

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.

Criteria for Success:

Now, your search agent should solve:
`-l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem`
`-l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem`

To receive full credit, 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).

Hint: 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.

My implementation of breadthFirstSearch expands just under 2000 search nodes on mediumCorners. However, heuristics (used with A* search) can reduce the amount of searching required.

You will also want to use the autograder with `-q q5` to verify everything is working.

Assignment 6 – Corners Problem Heuristic:

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

Admissibility vs. Consistency: Remember, heuristics are just functions that take search states and return numbers that estimate the cost to a nearest goal. More effective heuristics will return values closer to the actual goal costs. To be admissible, the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal (and non-negative). To be consistent, it must additionally hold that if an action has cost c, then taking that action can only cause a drop in heuristic of at most c.

Admissibility isn't enough to guarantee correctness in graph search – you need the stronger condition of consistency. However, admissible heuristics are usually also consistent, especially if they are derived from problem relaxations. Therefore it is usually easiest to start out by brainstorming admissible heuristics. Once you have an admissible heuristic that works well, you can check whether it is indeed consistent, too. The only way to guarantee consistency is with a proof. However, inconsistency can often be detected by verifying that for each node you expand, its successor nodes are equal or higher in in f-value. Moreover, if UCS and A* ever return paths of different lengths, your heuristic is inconsistent. This stuff is tricky!

Non-Trivial Heuristics: The trivial heuristics are the ones that return zero everywhere (UCS) and the heuristic which computes the true completion cost. The former won't save you any time, while the latter will timeout the autograder. You want a heuristic which reduces total compute time, though for this assignment the autograder will only check node counts (aside from enforcing a reasonable time limit).

Criteria for Success:

Now, your search agent should solve:
`-l mediumCorners -p AStarCornersAgent -z 0.5`

Your heuristic must be a non-trivial non-negative consistent heuristic to receive any points. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll be graded:

0/3 pts for expanding more than 2000 nodes
1/3 pts for expanding at most 2000 nodes
2/3 pts for expanding at most 1600 nodes
3/3 pts for expanding at most 1200 nodes

You will want to use the autograder with `-q q6` to verify everything is working.

Assignment 7 – Eating All the Dots:

For this problem we want to eat all the Pacman food in as few steps as possible. We'll need a new search problem definition which formalizes the food-clearing problem: `FoodSearchProblem` in `searchAgents.py` (implemented for you). A solution is defined to be a path that collects all of the food in the Pacman world. For the present assignment, solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food and Pacman. (Of course ghosts can ruin the execution of a solution! We'll get to that in future labs.) If you have written your general search methods correctly, A* with a null heuristic (equivalent to uniform-cost search) should quickly find an optimal solution to testSearch with no code change on your part (total cost of 7):

```
-l testSearch -p AStarFoodSearchAgent
```

You should find that UCS starts to slow down even for the seemingly simple tinySearch so we really need a good heuristic.

Fill in foodHeuristic in `searchAgents.py` with a consistent heuristic for the FoodSearchProblem.

Criteria for Success:

Try your agent on the trickySearch board:

```
-l trickySearch -p AStarFoodSearchAgent
```

My UCS agent finds the optimal solution exploring over 16,000 nodes.

Any non-trivial non-negative consistent heuristic will receive 1 point. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll get additional points:

1/4 pts for expanding more than 15000 nodes
2/4 pts for expanding at most 15000 nodes
3/4 pts for expanding at most 12000 nodes
4/4 pts for expanding at most 9000 nodes
5/4 pts for expanding at most 7000 nodes (hard).

You will want to use the autograder with `-q q7` to verify everything is working and don't be alarmed if the tests take some time to run.

Assignment 8 – Suboptimal Search:

Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard. In these cases, we'd still like to find a reasonably good path, quickly. In this assignment, you'll write an agent that always greedily eats the closest dot.

`ClosestDotSearchAgent` is implemented for you in `searchAgents.py`, but it's missing a key function that finds a path to the closest dot.

Implement the function `findPathToClosestDot` in searchAgents.py

Hint: The quickest way to complete `findPathToClosestDot` is to fill in the class `AnyFoodSearchProblem`, which is missing its goal test. Then, solve that problem with an appropriate search function. The solution should be very short!

Your ClosestDotSearchAgent won't always find the shortest possible path through the maze. To make sure you understand why you can try to come up with a small example where repeatedly going to the closest dot does not result in finding the shortest path for eating all the dots.

Criteria for Success:

Test your function with:
`-l bigSearch -p ClosestDotSearchAgent -z .5`

You will also want to use the autograder with `-q q8` to verify everything is working.

Submit your code in Canvas for grading.