PROJECT 1

Course: CSE 537

Due date: September 21, 11:59pm

TEAM MEMBER 1:
Name: Ayushi Srivastava
Student Id: 112101239
Email: ayushi.srivastava@stonybrook.edu

TEAM MEMBER 2:
Name: Jahnavi Tharigopula
Student Id: 112078393
Email: jahnavi.tharigopula@stonybrook.edu

Finding a Fixed Food Dot using Search Algorithms

Question 1:

DFS

Commands:

1. python pacman.py -l tinyMaze -p SearchAgent

```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 10 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 500
Average Score: 500.0
Scores:        500.0
Win Rate:      1/1 (1.00)
Record:        Win
```

2. python pacman.py -l mediumMaze -p SearchAgent

```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 130 in 0.0 seconds
Search nodes expanded: 146
Pacman emerges victorious! Score: 380
Average Score: 380.0
Scores:        380.0
Win Rate:      1/1 (1.00)
Record:        Win
```

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

```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 390
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

As mentioned, the Pac-Man board shows 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?**
    - Yes.
    - Theoretically, DFS explores the deepest nodes first. This order of exploration is evident in the overlay of the states explored where the deepest nodes (the nodes towards the left of the board) are explored first (showing in dark red) and the redness of the nodes decreases as the Pacman reaches the goal.
- **Does Pac-Man actually go to all the explored squares on its way to the goal?**
    - No
    - The explored squares indicate all the nodes that have been in the Fringe list whereas the path explored consists of the nodes that are in the Expanded list. Not all the nodes in the fringe list are expanded, hence not all the explored nodes are in the final path to the goal.
- **Is this a least cost solution?**
    - No.
    - We can see that the path taken to reach the goal is not the optimal one. The Pacman goes to the leftmost side of the board (as they are the deepest nodes) despite there being a closer path to the goal without having to visit the left regions, hence showing that this is not the least cost solution.
- **If not, think about what depth-first search is doing wrong.**
    - DFS uses a stack (LIFO) for the fringe list, i.e it expands the most recently explored state and then a state in its successors, and so on, thus expanding upto the deepest *and leftmost* node first.
    - Consider the goal to be at the Rightmost part of the graph and an infinite no.of states exist in the left part of the graph. In this case, the DFS will continue to expand the nodes in the left part away from the goal, thus increasing the no.of nodes explored in the path and causing the solution to be non-optimal.

Question 2:

BFS

Commands:

1. python pacman.py -l tinyMaze -p SearchAgent -a fn=bfs

```
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 502
Average Score: 502.0
Scores:        502.0
Win Rate:      1/1 (1.00)
Record:        Win
```

2. python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

```
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

3. python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5

```
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

- **Does BFS find a least cost solution? If not, check your implementation.**
    - Yes, BFS finds the least cost solution for the current problem with unit cost paths.

We then ran the eightpuzzle which uses BFS.

Commands:

1. python eightpuzzle.py

```
After 11 moves: left
-------------
|   | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
```

- We find the optimal solution to the eightpuzzle problem without making any change to the above BFS code.

Question 3:

UCS

Commands:

1. python pacman.py -l tinyMaze -p SearchAgent -a fn=ucs

```
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 502
Average Score: 502.0
Scores:        502.0
Win Rate:      1/1 (1.00)
Record:        Win
```

2. python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

```
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.2 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

3. python pacman.py -l bigMaze -p SearchAgent -a fn=ucs -z .5

```
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.7 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Varying the Cost Functions:

We ran our UCS algorithm on the mediumDottedMaze and mediumScaryMaze which use different cost functions.

4. python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

```
Path found with total cost of 1 in 0.1 seconds
Search nodes expanded: 186
Pacman emerges victorious! Score: 646
Average Score: 646.0
Scores:        646.0
Win Rate:      1/1 (1.00)
Record:        Win
```

5. python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
   (Pacman moves very slowly)

```
Path found with total cost of 68719479864 in 0.1 seconds
Search nodes expanded: 108
Pacman emerges victorious! Score: 418
Average Score: 418.0
Scores:        418.0
Win Rate:      1/1 (1.00)
Record:        Win
```

- As mentioned, with the varying cost functions, we observer a very low (cost = 1) and very high (cost = 68719479864) path costs for the StayEastSearchAgent and the StayWestSearchAgents respectively.

Question 4:

A* Search

Commands:

1. python pacman.py -l tinyMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

```
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 14
Pacman emerges victorious! Score: 502
Average Score: 502.0
Scores:        502.0
Win Rate:      1/1 (1.00)
Record:        Win
```

2. python pacman.py -l mediumMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

```
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.1 seconds
Search nodes expanded: 221
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

A* on the bigMaze using the Manhattan distance heuristic:

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

```
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.4 seconds
Search nodes expanded: 549
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

UCS on the bigMaze (which doesn't consider any heuristic, equivalent to using the NULL heuristic):

4. python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=ucs

```
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.5 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

As mentioned, comparing A* and UCS on the bigMaze we see that A* performs better than UCS. For the bigMaze, A* expands 549 nodes while UCS expands 620 nodes.

Now, we analysed what happens on openMaze for the various search strategies?

Commands:

1. python pacman.py -l openMaze -p SearchAgent -a fn=dfs -z .5

```
[SearchAgent] using function dfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 298 in 0.1 seconds
Search nodes expanded: 808
Pacman emerges victorious! Score: 212
Average Score: 212.0
Scores:        212.0
Win Rate:      1/1 (1.00)
Record:        Win
```

   - This DOES NOT take an optimal path

2. python pacman.py -l openMaze -p SearchAgent -a fn=bfs -z .5

```
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.1 seconds
Search nodes expanded: 682
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
```

   - This takes an optimal path

3. python pacman.py -l openMaze -p SearchAgent -a fn=ucs -z .5

```
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.4 seconds
Search nodes expanded: 682
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
```

   - This also takes an optimal path

4. python pacman.py -l openMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

```
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.4 seconds
Search nodes expanded: 535
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
```

   - This takes an optimal path as well.

To summarize the above results for the 4 algorithms, the **No.of Nodes Expanded** (which can be considered to describe the Memory Usage) and the **Time Taken** are as follows for the different mazes.

	DFS			BFS			UCS			A*		
Maze	**Time**	**Nodes expanded**	**Total Cost**	**Time**	**Nodes expanded**	**Total Cost**	**Time**	**Nodes expanded**	**Total Cost**	**Time**	**Nodes expanded**	**Total Cost**
tiny Maze	0.0	15	10	0.0	14	8	0.0	15	8	0.0	14	8
medium Maze	0.0	146	130	0.0	269	68	0.2	269	68	0.1	221	68
big Maze	0.0	390	210	0.0	620	210	0.7	620	210	0.4	549	210
open Maze	0.1	808	298	0.1	682	54	0.4	682	54	0.4	535	54

Observations:

- We observe that DFS does not follow the optimal path, and always has the highest total path cost as compared to the cost that is obtained using BFS, UCS and A*.
- Among the A* and UCS solutions, similar to what was discussed previously we see that the no.of nodes expanded by A* is always lesser than the no.of nodes expanded by UCS proving that considering a heuristic function provides a more efficient solution in terms of the No.of Nodes expanded.
- UCS consider the cost of the actions and A* consider both the cost of the actions and the heuristic value before computing the path, and hence they take longer to complete as compared to BFS and DFS.

Finding All the Corners

Question 5:

Cornersproblem

Commands:

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

```
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 253
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
```

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

```
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 253
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
```

As mentioned, using BFS for the CornersProblems, our solution also expands just under **2000** (our solution = **1967**) search nodes on mediumCorners.

However, heuristics (used with A* search) can reduce the amount of searching required.

Problem Definition:

- Initialization
    - Nothing extra was initialized apart from the already initialized information.

- Get Start State
    - The state space representation used for this problem contains information of both the current state and the no.of corners that are unvisited by the time Pacman is in the current state.

- This information is stored as a tuple containing [state, unvisitedcorners] where ‘state’ is of the form (x, y) and ‘unvisitedcorners’ is a tuple of corners that have not been visited yet.
- So based on the above definition of the state space, we get the Start State as (self.startingPosition, self.corners) as self.corners contains information about the positions of the 4 corners.

- Is Goal State
  - The goal state is defined as the state when the Pacman has visited all the 4 corners. Hence, our problem returns True when the tuple of ‘unvisitedcorners’ in the state space is empty.
  - And, also if hypothetically the starting state itself is a goal state, i.e all the corners are already visited, our model returns True.

- Get Successors
  - We get the successor states to the current state and check if this is a Corner state or not.
  - In case the successor is a corner state, we remove this corner from the tuple of unvisitedcorners, else the unvisitedcorners tuple remains the same as that of the previous state.
  - And we assume the stepcost to be 1.
  - So, we append the list of triples containing the ((next_state, updated_unvisitedcorners), actions, 1) into the successors list and return the successor list.

Question 6:

CornersHeuristic

Commands:

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

```
Path found with total cost of 106 in 0.2 seconds
Search nodes expanded: 742
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

- For determining heuristic function , we started simple function where food heuristic = min of all Manhattan distances of corners from the agent. Some of the test cases were failing with this solution.
- Changed the algorithm to finding out all the permutations of the unvisited array and taking the minimum of all the Manhattan distances found.

Question 7:

FoodHeuristic

Commands:

1. python pacman.py -l testSearch -p AStarFoodSearchAgent

```
((2, 3), <game.Grid instance at 0x10e75eb00>)
Path found with total cost of 7 in 0.0 seconds
Search nodes expanded: 10
Pacman emerges victorious! Score: 513
Average Score: 513.0
Scores:        513.0
Win Rate:      1/1 (1.00)
Record:        Win
```

2. python pacman.py -l trickySearch -p AStarFoodSearchAgent

```
((9, 3), <game.Grid instance at 0x101ee4518>)
Path found with total cost of 60 in 74.6 seconds
Search nodes expanded: 4137
Pacman emerges victorious! Score: 570
Average Score: 570.0
Scores:        570.0
Win Rate:      1/1 (1.00)
Record:        Win
```

3. python pacman.py -l testSearch -p AStarFoodSearchAgent

```
((4, 3), <game.Grid instance at 0x1042c6d40>)
Path found with total cost of 27 in 11.4 seconds
Search nodes expanded: 2372
Pacman emerges victorious! Score: 573
Average Score: 573.0
Scores:        573.0
Win Rate:      1/1 (1.00)
Record:        Win
```

- For determining food heuristic function , we started out with a simple function where food heuristic = sum of all Manhattan distances of food from the agent. Although the expanded nodes were less than 6000, but the function was inconsistent.
- So we changed the algorithm to finding out the max Manhattan distance between food and agent. The algorithm was giving heuristic value to the goal not 0, which was wrong.
- Finally, tested with the maze distance between agent and food. Although it takes longer time to execute but the search nodes expanded are around 4137. It gives a consistent heuristic function as the maze distance always returns the same distance from food to agent, so we don't have to find the max distance from food to agent as in previous case.
- Our heuristic is admissible and is a monotonic function.