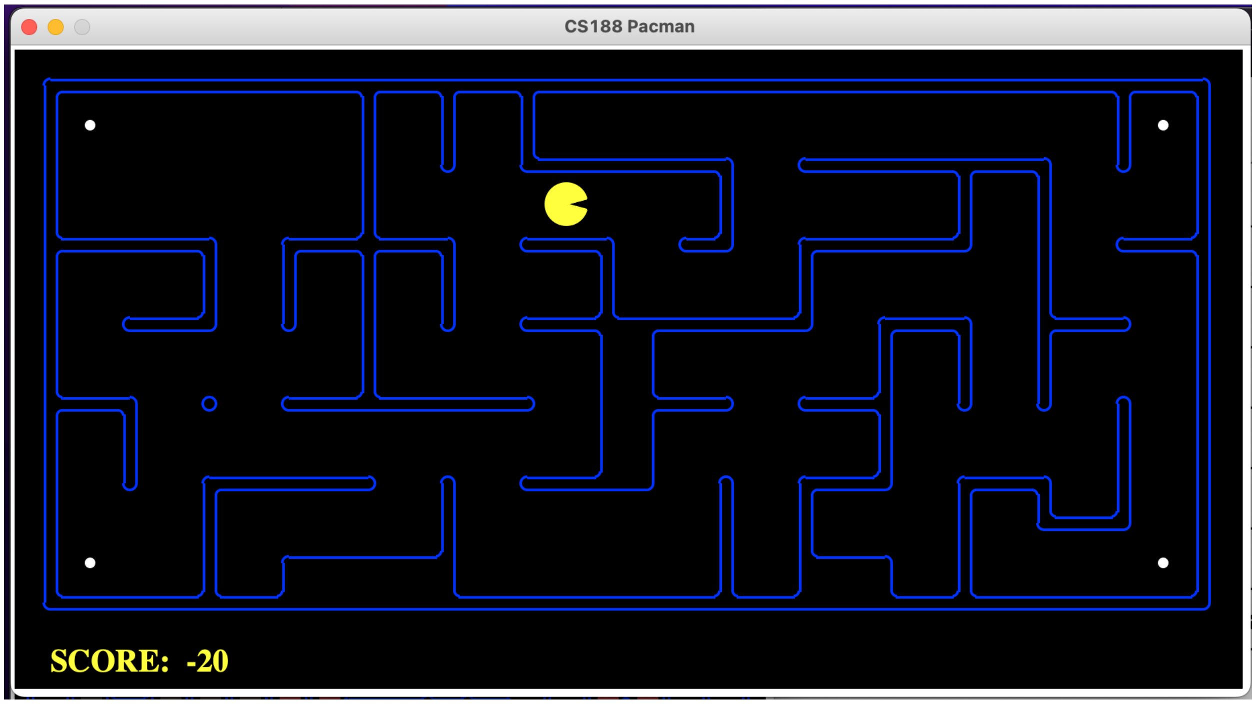
Project 01
The Searchin' Pac-Man

By

Group 26
Akhilesh Boppana (SBU ID: 114841021)
Jugal Deepak Chauhan (SBU ID: 115059432)

CONTENTS

1. Introduction



Pac-Man is a 90's maze action video game. The gamer playing the game controls the PacMan (yellow circle with a mouth) and can give commands to the PacMan to move in four directions - North, East, West & South. The objective of this game is to eat all the dots lying around in the Maze while avoiding contact with ghosts. If PacMan eats all the dots in a particular maze, it wins.

2. Problem Statement

In this project, the PacMan Agent navigates a maze-like world to reach specific locations and find efficient ways to gather food. Implementation requires using common search algorithms such as depth-first, breadth-first, uniform cost & A* search algorithms which help in solving navigation problems in the world of PacMan.

This assignment consists of seven questions which at every step, provide insinuating hints which guide in trying to develop an optimum solution. The questions also indicate which particular function needs to be changed and provides respective python commands to run and test the code.

3. Project File Description

search.py: Where all the search algorithms are located
searchAgents.py: Where all the search-based agents are located
pacman.py: It is the main file that runs the PacMan game. It also describes different types of GameState
game.py: Stores the logic behind the PacMan game world
util.py: Stores useful information and data structures for implementing search algorithms

Some other supporting files -
graphicsDisplay.py: Graphics for Pacman.
graphicsUtils.py: Support for Pacman graphics.
textDisplay.py: ASCII graphics for Pacman.
ghostAgents.py: Agents to control ghosts.
keyboardAgents.py: Keyboard interfaces to control Pacman.
layout.py: Code for reading layout files and storing their contents.
autograder.py: Project autograder.
testParser.py: Parses autograder test and solution files.
testClasses.py: General auto grading test classes.
test_cases/: Directory containing the test cases for each scenario.
searchTestClasses.py: Project specific auto grading test classes.

4. Questions

The following seven sub-points will provide a summary of our solution to the given problems including different data structures, outputs & statistics.

4.1 Question 1: Depth First Search

A depth-first search always expands the deepest node within the current limits of the search tree. It uses a LIFO *(last in first out)* Queue - which is also known as a Stack. A Stack means that the most recently generated node is most suitable for expansion. In fact, the solution returned by the algorithm may not contain all expanded nodes, so Pac-Man does not have to go through every square on the way to the goal.
Unfortunately, DFS does not promise an optimal solution. This is because we know this solution is not the most cost-effective solution.

Data Structures:

Stack - It maintains the current state and path till the current state (Used as a fringe list with LIFO)

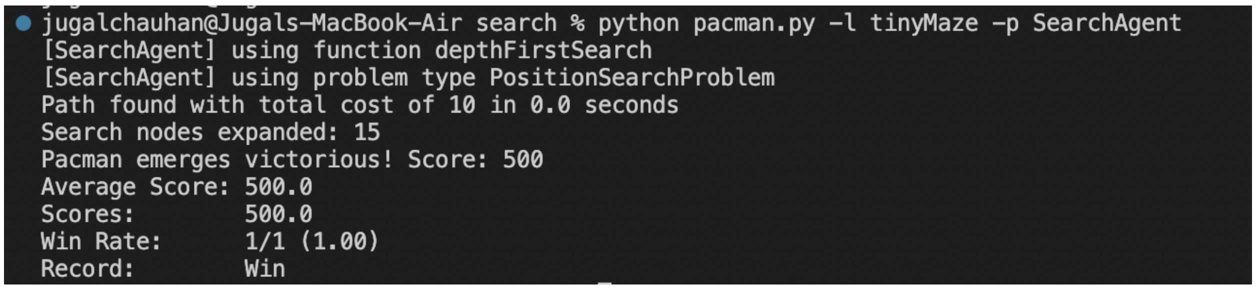
Visited - It is a Dictionary containing the nodes which have been popped from the stack, and the corresponding action.

Time Complexity: O(b^m) [here b = 3]

Memory Usage (Space Complexity)**:** O(bm) [here b = 3]
(b = branching factor i.e. number of successors of each node m = maximum depth of solution)

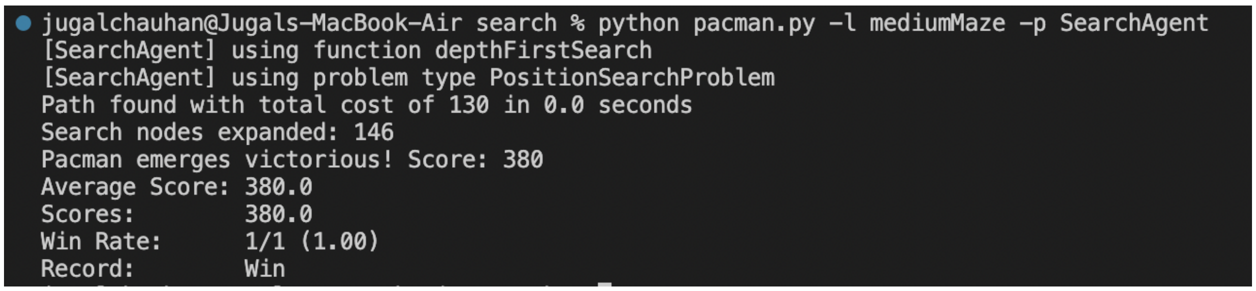
Outputs :

python pacman.py -l tinyMaze -p SearchAgent



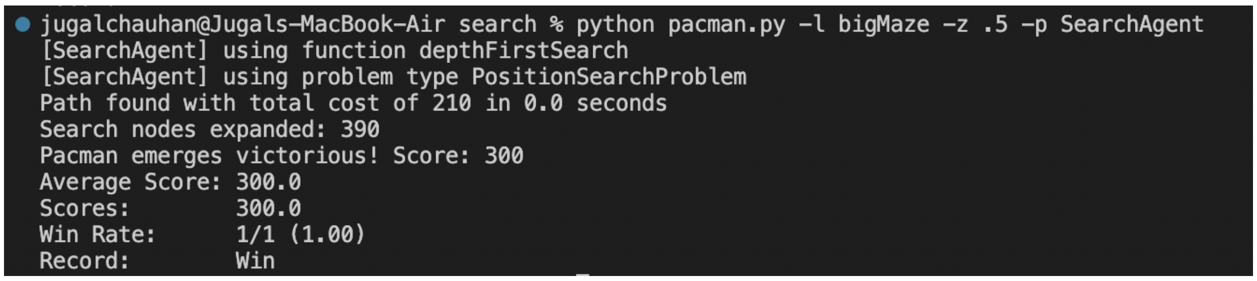
```
● jugalchauhan@Jugals-MacBook-Air search % python pacman.py -l tinyMaze -p SearchAgent
  [SearchAgent] using function depthFirstSearch
  [SearchAgent] using problem type PositionSearchProblem
  Path found with total cost of 10 in 0.0 seconds
  Search nodes expanded: 15
  Pacman emerges victorious! Score: 500
  Average Score: 500.0
  Scores:        500.0
  Win Rate:      1/1 (1.00)
  Record:        Win
```

python pacman.py -l mediumMaze -p SearchAgent



```
● jugalchauhan@Jugals-MacBook-Air search % 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
```

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



```
● jugalchauhan@Jugals-MacBook-Air search % 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
```

Statistics :

Maze Type	Total Cost	Nodes Expanded	Score	Win Rate
tinyMaze	10	15	500	1.0
mediumMaze	130	146	380	1.0
bigMaze	210	390	300	1.0

Critical Analysis:

The exploration order is as expected as per the DFS algorithm.
Does Pac-Man actually go to all the explored squares on its way to the goal?
No, It does not. In any path, if the Pacman finds a goal state. It would just stop there, there might be other states on the fringe. Therefore, all the states are not explored

4.2 Question 2: Breadth First Search

Breadth-first search is a simple strategy that first expands the root node, then all children of the root node, then their children. In general, all nodes at a certain depth in the search tree are expanded before the next level nodes are expanded. Whereas DFS uses a LIFO queue, BFS is implemented simply by using a FIFO *(first in first out)* queue (which is also called Queue).
Breadth First Search returns a least-cost solution in the sense of how many actions it takes for PacMan to reach the food dot.

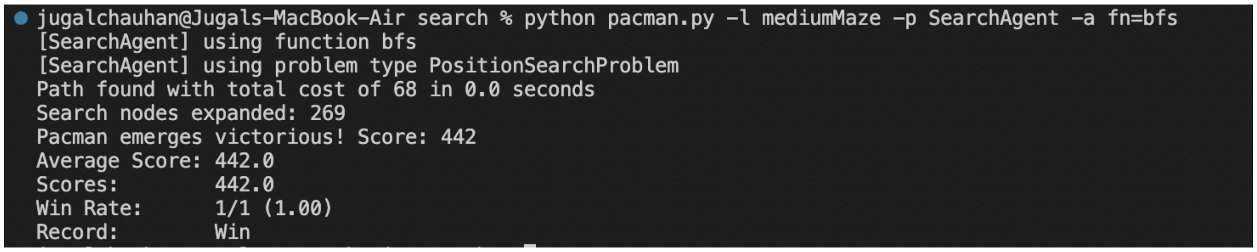
Data Structures:
Queue - fringe (Same as DFS except the fringe list is a queue which follows FIFO)
visited - List maintaining the states which are visited and expanded already.

Time Complexity: O(b^{d+1}) [here b = 3]

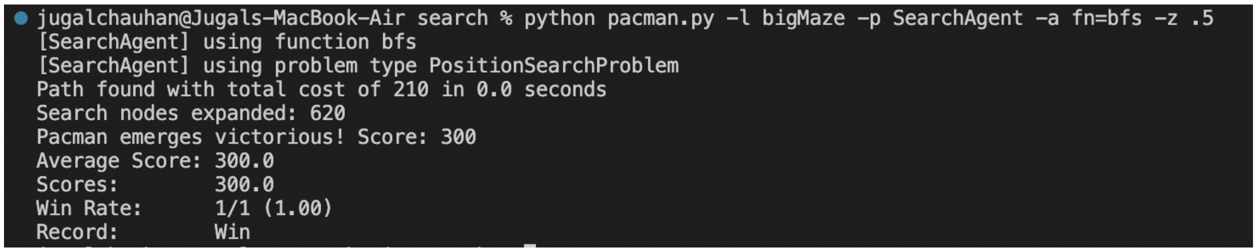
Memory Usage (Space Complexity)**:** O(b^{d+1}) [here b = 3]

(b = branching factor i.e. number of successors of each node d = depth of optimal solution)

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


```
jugalchauhan@Jugals-MacBook-Air search % 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
```

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


```
jugalchauhan@Jugals-MacBook-Air search % 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
```

Statistics:

Maze Type	Total Cost	Nodes Expanded	Score	Win Rate
mediumMaze	68	269	442	1.0
bigMaze	210	620	300	1.0

4.3 Question 3: Uniform Cost Search

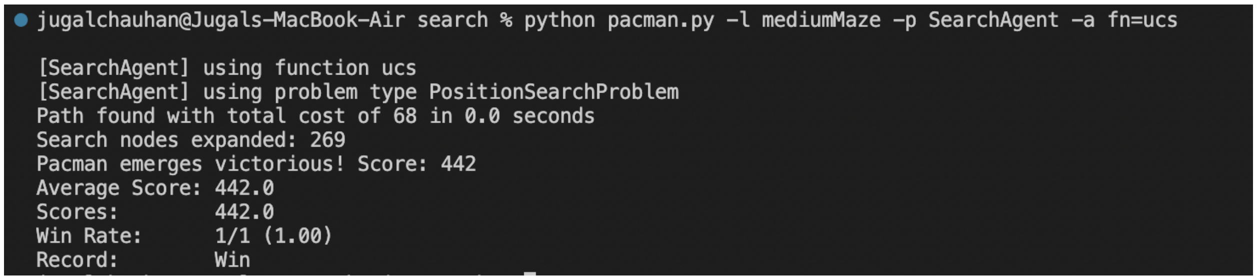
Instead of expanding the obvious node, UCS expands the node with the lowest path cost. This is done by storing the values in a cost-ordered priority queue. In this particular Pac-Man scenario, the cost function allows us to take into account the risk of getting caught by ghosts and the chances of getting more food.

Data Structures:
Priority Queue
visited - List maintaining the states which are visited and expanded already.
Time Complexity: $O(b^{1 + \lfloor C^* / \epsilon \rfloor})$ [here b = 3, ϵ = 1]

Memory Usage (Space Complexity):$O(b^{1 + \lfloor C^* / \epsilon \rfloor})$ [here b = 3, ϵ = 1] (b = branching factor i.e. number of successors of each node
C^* = cost of optimal solution
ϵ = maximum cost of each action)

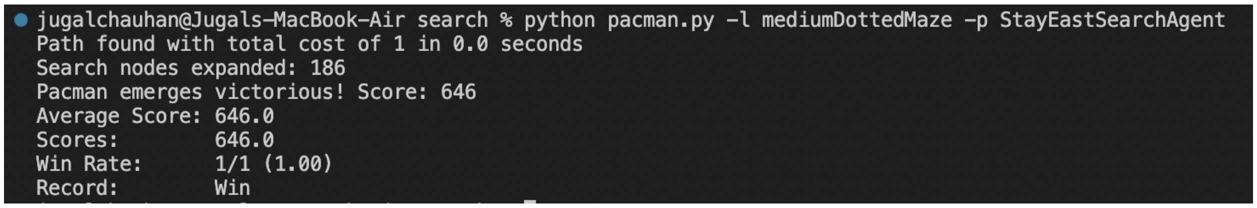
Outputs :

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



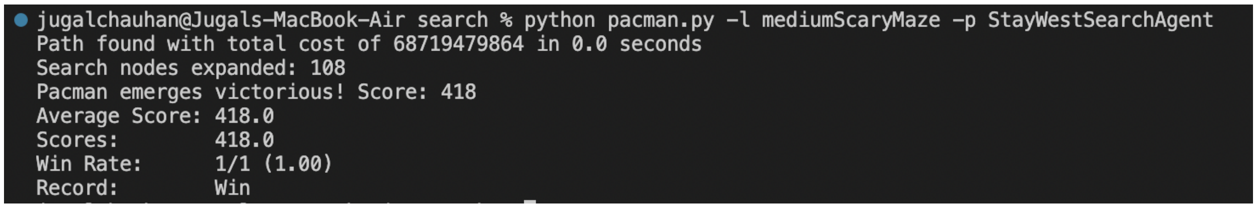
```
jugalchauhan@Jugals-MacBook-Air search % python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 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
```

python pacman.py -l mediumDottedMaze -p StayEastSearchAgent



```
jugalchauhan@Jugals-MacBook-Air search % python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
Path found with total cost of 1 in 0.0 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
```

python pacman.py -l mediumScaryMaze -p StayWestSearchAgent



```
jugalchauhan@Jugals-MacBook-Air search % python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
Path found with total cost of 68719479864 in 0.0 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
```

Statistics :

Maze Type	Total Cost	Nodes Expanded	Score	Win Rate
mediumMaze	68	269	442	1.0
mediumDottedMaze	1	186	646	1.0
mediumScaryMaze	68719479864	100	418	1.0

4.4 Question 4 : A* Search

One of the most common forms of best-first search is called an A* search. Evaluate a node by combining g(n), the cost to reach the node, and h(n), the cost from the node to the destination: f(n) = g(n) + h(n). Implemented A* search to speed up the search. Here we use the already implemented Manhattan distance as a heuristic function and want to observe the search speed improvement.

Data Structures:
Priority Queue
visited - List maintaining the states which are visited and expanded already.

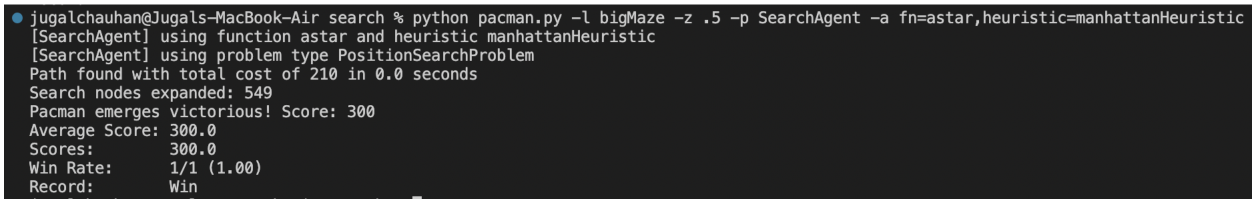
Time Complexity: $O(b^d)$ [here b = 3]

Memory Usage (Space Complexity):$O(b^d)$ [here b = 3]

(b = branching factor i.e. number of successors of each node d = depth of optimal solution)

This is the worst case complexity obtained when null heuristic is used(same as BFS). It changes according to the heuristic used.

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



```
● jugalchauhan@Jugals-MacBook-Air search % 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 0.0 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
```

Statistics : Comparison between A* and USC

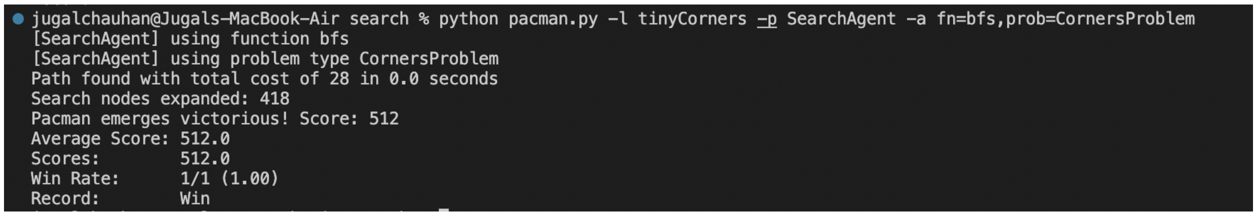
<u>Algorithm</u>	<u>Maze Type</u>	<u>Total Cost</u>	<u>Nodes Expanded</u>	<u>Score</u>
A*	bigMaze	210	549	300
USC	bigMaze	210	620	300

4.5 Question 5: Corner’s Problem

This part of the project aims to make discrepancies between search methods more explicit. The true power of the A* search becomes clearer in more sophisticated search problems. Implemented a corner problem where the objective of the quest is to eat 4 feed points at each corner of the maze instead of eating one with the shortest path.

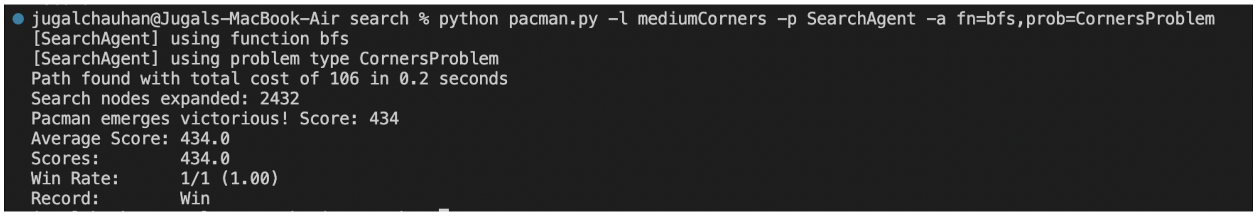
Outputs :

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



```
jugalchauhan@Jugals-MacBook-Air search % 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: 418
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
```

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



```
jugalchauhan@Jugals-MacBook-Air search % 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 106 in 0.2 seconds
Search nodes expanded: 2432
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Statistics :

Maze Type	Total Cost	Nodes Expanded	Score	Win Rate
tinyMaze	28	418	512	1.0
mediumMaze	106	2432	434	1.0

4.6 Question 6: Heuristic for Corner's Problem

We have devised our own heuristic function serving to save more time while searching. Since our problem here is a graph search problem, we need to design a heuristic not only admissible but also consistent.

Heuristic:

1. Initialize the heuristic to zero
2. The Maze distances between Pacman's current position and all the remaining corners are calculated
3. The corner having maximum Maze distance is chosen, and this distance is added to the heuristic.
4. The heuristic is returned.

Admissibility:

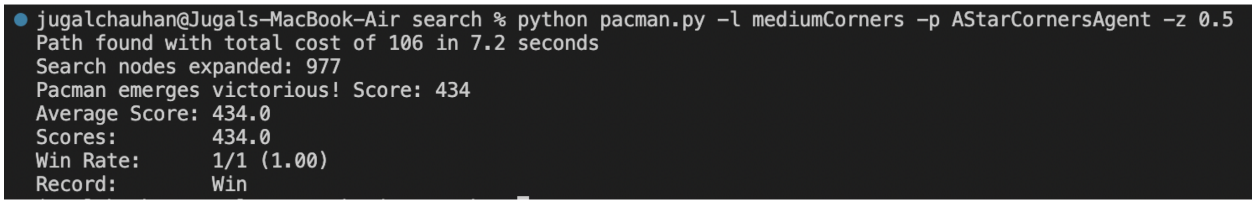
The heuristic is admissible because in any case, Pacman must cover the distance from the current position to the closest corner, and also the distance from the closest corner to the farthest corner from there. (There are no exceptions.) So, we could have chosen the nearest corner distance, but the farthest corner distance is always greater than the nearest corner distance (So, the farthest distance heuristic dominates the nearest distance heuristic).

Consistency:

The heuristic is consistent because as Pacman reaches closer to the corner, the Maze distance between the current position and the closest corner keeps decreasing and reaches zero when the goal state is reached.

Outputs :

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



```
● jugalchauhan@Jugals-MacBook-Air search % python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
  Path found with total cost of 106 in 7.2 seconds
  Search nodes expanded: 977
  Pacman emerges victorious! Score: 434
  Average Score: 434.0
  Scores:        434.0
  Win Rate:      1/1 (1.00)
  Record:        Win
```

Statistics :

Type	Total Cost	Nodes Expanded	Score	Win Rate
mediumCorners	106	977	434	1.0

4.7 Question 7 : Eating all the dots

Heuristic :

1. Initialize the heuristic to zero
2. The Maze distances between Pacman's current position and all the remaining food are calculated
3. The food having maximum Maze distance is chosen, and this distance is added to the heuristic.
4. The heuristic is returned.

Admissibility:

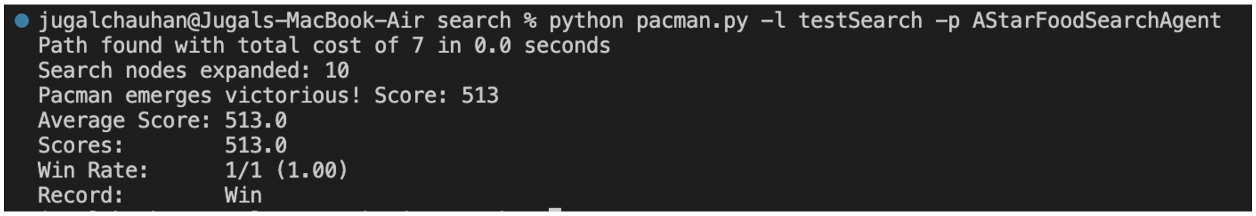
The heuristic is admissible because in any case, Pacman must cover the distance from the current position to all the food dots. So, we could have chosen the nearest food distance, but the farthest food distance is always greater than the nearest food distance for all positions(So, the farthest distance heuristic dominates the nearest distance heuristic).

Consistency:

The heuristic is consistent because as Pacman reaches closer to the food, the Maze distance between the current position and the closest corner keeps decreasing and eventually reaches to zero when the goal state is reached.

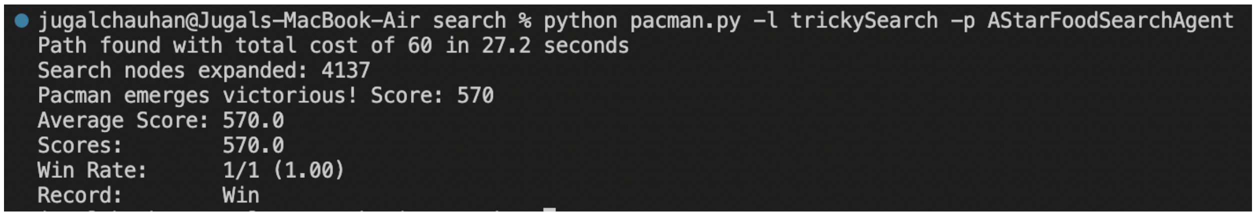
Outputs :

python pacman.py -l testSearch -p AStarFoodSearchAgent



```
● jugalchauhan@Jugals-MacBook-Air search % python pacman.py -l testSearch -p AStarFoodSearchAgent
  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
```

python pacman.py -l trickySearch -p AStarFoodSearchAgent



```
● jugalchauhan@Jugals-MacBook-Air search % python pacman.py -l trickySearch -p AStarFoodSearchAgent
  Path found with total cost of 60 in 27.2 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
```

Statistics :

Search Type	Total Cost	Nodes Expanded	Score	Win Rate
testSearch	7	10	513	1.0
trickySearch	60	4137	570	1.0

6. Other efforts and Conclusion:

Before coming up with this heuristic, we tried and implemented various other heuristics:

1. Euclidean distance between the current position and closest food/corner.

2. Euclidean distance between the current position and farthest food/corner.

3. Manhattan distance between the current position and closest food/corner.

4. Manhattan distance between the current position and farthest food/corner.

Although, all the above heuristics were admissible. The current one using maze distance gave the best optimality. As it was a dominant heuristic when compared to all others listed above.