Assignment1. Search in Pacman Project Report

Shihan Ran - 15307130424

***Abstract*—This project is aimed at designing a intelligent Pacman agent that is able to find optimal paths through its maze world considering both reaching particular locations (e.g., finding all the corners) and eating all the dots in as few steps as possible. It can be separated as two subtasks: implementing graph search algorithms for DFS, BFS, UCS as well as A*, and use the search criteria outlined in the lectures to design effective heuristics.**

In terms of the first task, since each algorithm is very similar, all algorithms differ only in the details of how the fringe is managed. Hence I implemented a single generic method which is configured with an algorithm-specific queuing strategy and apply them to Pac-man scenarios. During this project, alone with implementing the already well-framed code block, I've spent much time improving my code's efficiency and comparing different implementation of heuristics.

I. Introduction

PAC-MAN is one of the most popular game in the world. The goal of this game is to accumulate points by eating all the Pac-Dots in the maze, completing that 'stage' of the game and starting the next stage and maze of Pac-dots. There are several ghosts roaming the maze, trying to kill Pac-Man. If any of the ghosts hit Pac-Man, he loses a life; when all lives have been lost, the game is over. Its interface is like figure 1.

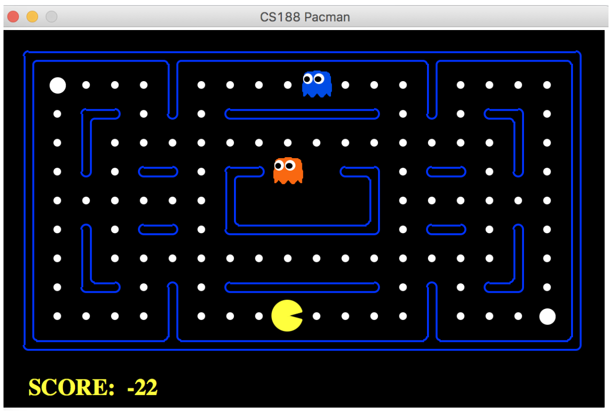


Fig. 1. The interface of Pac-man.

With the help of problem-solving agent, we can automatically find optimal paths through its maze world considering both reaching particular locations (e.g., finding all the corners) and eating all the dots in as few steps as possible. To design this intelligent agent, we implemented several uninformed search algorithms-algorithms that are given no information about the problem other than its definition (e.g., DFS, BFS, UCS). However, although some of these algorithms can solve any solvable problem, none of them can do so efficiently. Consequently, we introduced informed search algorithms (e.g., A*), which, on the other hand, can do quite well given some guidance on where to look for solutions.

For the following parts of the report, we will begin with the introduction of generic graph search algorithms firstly, explaining the principles of search algorithms and point out the differences in algorithm-specific queuing strategy. Then we turn to discuss informed search algorithms and the search criteria outlined in the lectures which we use to design effective heuristics.

II. Generic Search

The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand. The general **TREE-SEARCH** algorithm is shown informally in Figure 2. Actually, search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next-the so-called search strategy. In other words, algorithms for DFS, BFS, UCS, and A* differ only in the details of how the frontier is managed.

A. *Question 1: Depth First Search*

Depth-first search always expands the deepest node in the current frontier of the search tree. As I mentioned before, the depth-first search algorithm is an instance of the graph-search algorithm in

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state then return the corresponding solution
      add the node to the explored set
      expand the chosen node, adding the resulting nodes to the frontier
         only if not in the frontier or explored set
```

Fig. 2. An informal description of the general graph-search algorithms.

Figure 2; it uses a LIFO queue (which is also known as **Stack**). A LIFO queue means that the most recently generated node is chosen for expansion.

TABLE I
RESULTS OF DEPTH FIRST SEARCH

Maze Form	Total Cost	Nodes expanded	Score
tinyMaze	10	15	500
mediumMaze	130	146	380
bigMaze	210	390	300

Here we implement the depth first search algorithm and results can be shown as Table I. Also, Pac-man need not in fact bother to go to all the squares on his way to the goal as all the nodes expanded may not be in the solution returned by the algorithm. Unfortunately, DFS does not promise us the optimal solution as we see that the solution is not the least cost solution.

B. *Question 2: Breadth First Search*

Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. Whereas DFS uses a LIFO queue, BFS is achieved very simply by using a FIFO queue (which is also called **Queue**) for the frontier.

TABLE II
RESULTS OF BREADTH FIRST SEARCH

Maze Form	Total Cost	Nodes expanded	Score
tinyMaze	8	15	502
mediumMaze	68	269	442
bigMaze	210	620	300

Breadth First Search returns a least cost solution in the sense of how many actions it takes for Pacman to reach the food dot and results are shown in Table II. However, the numbers of expanded nodes are relatively large, reaches 269 and 620 for mediumMaze and bigMaze respectively, which costs a lot of time to find the optimal solution. My BFS algorithm also applies to the eight-puzzle search algorithm.

C. *Question 3: Varying the Cost Function*

When all step costs are equal, BFS is optimal because it always expands the shallowest unexpanded node. However, instead of expanding the shallowest node, UCS expands the node with the lowest path cost. This is done by storing the frontier as a **priority queue** ordered by cost. In this specific Pac-man scenarios, the cost functions enable us to take the perils of being caught by ghosts as well as the chance of getting more food into consideration.

TABLE III
RESULTS OF UNIFORM COST SEARCH

Maze Form	Total Cost	Nodes expanded	Score
mediumMaze	68	269	442
mediumDottedMaze	1	186	646
mediumScaryMaze	68719479864	108	418

We use priority queue as the frontier to pop out least cost node every time, which can promise an optimal solution if we use the cost from initial state to current state as our cost function. Results can be concluded as Table III.

I noted that we get very low and very high path costs for *StayEastSearchAgent* and *StayWestSearchAgent* respectively, due to their exponential cost functions. As the cost function for stepping into (x, y) for *stayEastSearchAgent* is $g = 0.5x$, the corresponding path cost would be pretty low. On the other hand, the final cost for the path given by *stayWestAgent* is 68719479864, which is very high because its cost function is $g = 2x$ for (x, y).

D. *Question 4: A* Search*

One of the most widely known form of best-first search is called A* search. It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal: $f(n) = g(n) + h(n)$. Since $g(n)$ gives the path cost from the start node to node n, and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have $f(n) =$ estimated cost of the cheapest solution through n.

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of $g(n) + h(n)$. The algorithm is identical to UCS except that A* uses $g + h$ instead of g.

TABLE IV
DIFFERENCES BETWEEN UCS AND A*

Algorithm	Maze Form	Total Cost	Nodes expanded
UCS	*mediumMaze*	68	269
A*	*mediumMaze*	68	**221**
UCS	*bigMaze*	210	620
A*	*bigMaze*	210	**549**

We now implement the A* search to boost the speed of searching. Here we use the already implemented Manhattan distance as the heuristic function and wish to observe an increase in the speed of searching. To better study the difference between UCS and A*, we listed total cost and search nodes expanded in Table IV. It turns out that this strategy is more than just reasonable: provided that the heuristic function $h(n)$ satisfies certain conditions, A* search is both complete and optimal.

For *openMaze* problem, we got results like Table V.

TABLE V
RESULTS OF OPENMAZE

Algorithm	Maze Form	Total Cost	Nodes expanded
DFS	*openMaze*	298	576
BFS	*openMaze*	54	682
UCS	*openMaze*	54	682
A*	*openMaze*	54	**535**

III. CORNERS PROBLEM

A. *Question 5: Finding All the Corners*

This part of the project intends to make the discrepancies between searching methods more obvious: the real power of A* Search will be more apparent with a more challenging search problem. So, we establish the CornersProblem, where the goal of the search is eating four food dots at each corner of the maze instead of eating one with the shortest path in previous questions.

One major problem here is how to detect whether all four corners have been reached (in other words, whether Pac-man achieves the goal state) in an abstract state representation. We finally decided to represent these states properly by a nested tuple like ((x,y), (corners to be visited)), where ”corners to be visited” is also a nested tuple representing which corners are left to be visited by the agent. And (x,y) is a tuple representing the position of this state. For example, ((3, 6), ((1, 1), (1, 6), (6, 6))) means Pac-man is now at (3,6), and the lower left corner, the lower right corner as well as the upper right corner are still not visited.

TABLE VI
RESULTS OF CORNERS PROBLEM USING BFS

Maze Form	Total Cost	Nodes expanded	Score
tinyCorners	28	252	512
mediumCorners	106	1966	434
bigCorners	162	7949	378

After this being done, we now implement BFS and A* Search to see the difference. Results of BFS are shown in the Table VI. We will implement A* Search in the next question.

B. *Question 6: Corners Problem: Heuristics*

Here we devise 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**. Finally we've come up with two different heuristics.

*1) **Heuristics 1:*** We've already known that admissible heuristics are usually also consistent, especially if they are derived from problem relaxations. Hence let's consider generating admissible heuristics from relaxed problems.

If we ignore all walls in the map, which is trivial in designing admissible heuristics. By doing this, the cost of walking from a position to another is the *Manhattan distance* of these two positions. Then for all corners unvisited, according to our definition, corner with the largest Manhattan Distance is the last one to be visited. After visiting this corner, our game reaches goal state.

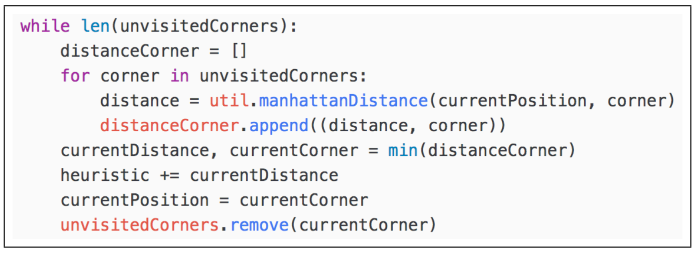
In this case, we simply use the largest Manhattan Distance as our heuristics value. It's obviously admissible, since it prevents overestimating the cost to reach the goal. And it's also consistent.

*2) **Heuristics 2:*** The idea of constructing this heuristics takes steps. Actually, we've inspired by heuristics 1.

First, we still use the Manhattan distance of two positions and ignore all walls in the map like what we did before.

Second, consider that if we have already reached a corner, then we only need to walk along the borders of the map to touch all unvisited corners. The most ideal condition is that we walk along the short side of the map first and touch the second unvisited corner, then walk along the long side of the map to touch the third unvisited corner, finally we walk along the short side of the map to touch the last unvisited corner. We assume this condition always happens to make our heuristics admissible, so to calculate the heuristics value of this part, we only need to concern the number of unvisited corners instead of their positions.

Third, we assume that we always firstly walk to the closest unvisited corner then take the second step above, so we only need to calculate the Manhattan distances between current position and unvisited corners to find the smallest distance to add to the heuristic value. After update heuristic value, we replace current position as the closest unvisited corner and do it again, until all corners are visited. The algorithm can be concluded like Figure 3.



```
while len(unvisitedCorners):
    distanceCorner = []
    for corner in unvisitedCorners:
        distance = util.manhattanDistance(currentPosition, corner)
        distanceCorner.append((distance, corner))
    currentDistance, currentCorner = min(distanceCorner)
    heuristic += currentDistance
    currentPosition = currentCorner
    unvisitedCorners.remove(currentCorner)
```

Fig. 3. Algorithm of Heuristics 2

We can find that all three steps above are admissible, so that our heuristic function is admissible. To prove it is consistent, its cost is calculated by the Manhattan distance so it is consistent.

*3) **Results**:* We concluded it as Table VII. Clearly, heuristics 2 is more effective than heuristics 1 since it will return values closer to the actual goal costs.

TABLE VII
RESULTS OF CORNERS PROBLEM USING DIFFERENT HEURISTICS

Heuristics	Maze Form	Total Cost	Nodes expanded
Heuristics 1	*mediumCorners*	106	1136
Heuristics 2	*mediumCorners*	106	**692**
Heuristics 1	*bigCorners*	162	4380
Heuristics 2	*bigCorners*	162	**1725**

IV. EATING FOOD PROBLEM

A. *Question 7: Eating All the Dots*

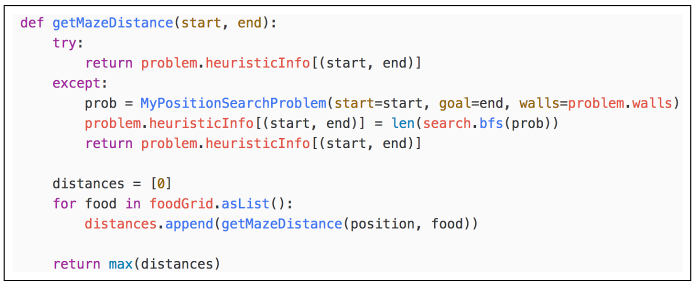
To make matters even more complicated, Pac-man now needs to eat all the dots in the world instead of only reaching the corners. Here the effect of A* Search becomes stronger. The same as what we did in the last problem, we too came up with several heuristic.

*1) **Heuristics 1**:* It's pretty similar with the one in corners problem, except for replacing corner list with food list.

We still ignore all walls and calculate *Manhattan distance* between current position to each dot and the one with the largest *Manhattan distance* is the last one to be visited. After eating this dot, our game reaches goal state.

In this case, we simply use the largest Manhattan Distance as our heuristics value. Admissible and consistent have been proofed before.

*2) **Heuristics 2**:* We can't simply follow the way in corners problem since there are too many dots in this case.

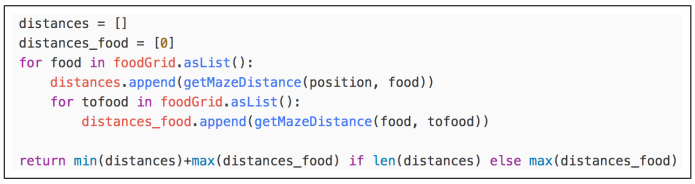


```
def getMazeDistance(start, end):
    try:
        return problem.heuristicInfo[(start, end)]
    except:
        prob = MyPositionSearchProblem(start=start, goal=end, walls=problem.walls)
        problem.heuristicInfo[(start, end)] = len(search.bfs(prob))
        return problem.heuristicInfo[(start, end)]

    distances = [0]
    for food in foodGrid.asList():
        distances.append(getMazeDistance(position, food))

    return max(distances)
```

Fig. 4. Algorithm of Heuristics 2

A reasonable idea is replacing *Manhattan distance* with other distances function that can find path more accurate. Hence a natural way of doing it is using the search functions you have already built to calculate the maze distance between any two points. The algorithm can be concluded like Figure 4. We also use *problem.heuristicInfo* to **store information** to be reused in other calls to the heuristic.

*3) **Heuristics 3**:* We use the shortest distance between Pac-man and food along with the longest distance between foods and foods. Algorithm can be simply concluded as Figure 5. It's consistent.



```
distances = []
distances_food = [0]
for food in foodGrid.asList():
    distances.append(getMazeDistance(position, food))
    for tofood in foodGrid.asList():
        distances_food.append(getMazeDistance(food, tofood))

return min(distances)+max(distances_food) if len(distances) else max(distances_food)
```

Fig. 5. Algorithm of Heuristics 3

*4) **Heuristics Others**:* In order to solve medium search, I also designed several other heuristic, unfortunately it's all proved to be inconsistent.

*5) **Results**:* We concluded it as Table VIII. Clearly, heuristics 2 outperforms heuristics 1 since it will return values closer to the actual goal costs.

TABLE VIII
RESULTS OF FOOD PROBLEM USING DIFFERENT HEURISTICS

Heuristics	Maze Form	Nodes expanded	Time
Heuristics 1	*trickySearch*	9551	4.2
Heuristics 2	*trickySearch*	4137	2.2
Heuristics 2	*trickySearch*	**721**	**2.1**

B. *Question 7.5: Solve medium search*

In the previous question, we were asked to write an *A* search heuristic* that would find the shortest path which visits every food dot at least once. This is obviously similar to the famously **NP-Hard Traveling Salesman Problem**, but the previous mazes we were required to solve were small enough that even an exponential solution would work. Also, the mazes were all *sparse* - most of the maze was empty, with only a handful of food dots, meaning we can speed things up by *only considering the food*. Instead of trying to find the shortest path in a large maze of n nodes, we can instead pretend to solve a problem in a maze of food nodes.

However, when it comes to *medium search*, old methods were incapable to solve it. For the *medium search* by contrast, the search space is exponentially large. Even though the maze itself is small, each possible subset of eaten food represents a different state in the search space and every square had food in it. In order to solve it, one must have a more sophisticated approach.

C. *Question 8: Suboptimal Search*

The problem indicates that even a good heuristic would fail to find the optimal path in a short time. Being such the case, its more realistic to find a reasonably good path, though not as good as the optimal path, in a relatively short time.

One of such agent is one which always eats the dot closest to Pac-man. The goal state test of *AnyFoodSearchProblem* can also be done by the already defined *gameState.getFood()* function, and the function of *findPathToClosestDot* can be simply constructed by search algorithms (e.g. BFS, UCS, A*) we implemented before. The results can be shown like Table IX.

TABLE IX
DIFFERENCES BETWEEN SEARCH ALGORITHMS

Algorithm	Maze Form	Total Cost	Score
DFS	*bigSearch*	5324	-2614
BFS	*bigSearch*	350	2360
UCS	*bigSearch*	350	2360
A*	*bigSearch*	350	2360

Nevertheless, the algorithm here can only be referred to as suboptimal, not optimal because it is a simply greedy search.