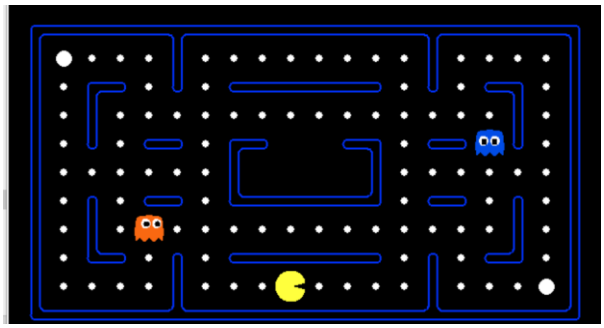
Project1: Search in Pacman

1. Introduction

Consider a Pacman as a search agent, teach a Pacman to find a path to his success. The version of the game is shown below.



The map is a table of positions and the Pacman can move by four directions (NSEW). Pacman's success differs from reaching particular locations to eating all the dots. Generally, it can be divided into two parts: search algorithms such as DFS, BFS, UCS and A*, effective heuristics for specific problem. (In code framework, search.py and searchAgents.py)

Search algorithms are a easy part due to their similarity. All algorithms differ only in the details of how the fringe is managed. Designing effective heuristics is a tricky part, we need design effective and consistent heuristics.

By the way, this is the first time that I confront OOT (object oriented technology) in python. I spend some time on understanding it.

2. Search algorithms

The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand. In order to avoid expanding any already visited states.

A. Finding a Fixed Food Dot using Depth First Search

Depth First Search always expands the deepest node in the current frontier. In order to accomplish the task, use a LIFO queue (data structure: Stack which is provided in utils.py). Here we implement DFS search results are shown below.

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

However, we can figure out that DFS is not optimal. It finds the "leftmost" solution, regardless of depth or cost.

B. Breadth First Search

Breadth First Search starts at the tree root , and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. In order to accomplish the task, use a FIFO queue (data structure: Queue which is provided in utils.py). Here we implement DFS search results are shown below.

Maze	Total Cost	Nodes expanded	Score
tinyMaze	8	15	502
mediumMaze	130	267	442
bigMaze	210	617	300

Compares to DFS, BFS finds optimal solution due to less total cost, but the number of expanded nodes is relatively large. My BFS algorithm also applies to the eight-puzzle search algorithms.

C. Varying the Cost Function

In this part, I need accomplish Uniform Cost Search (UCS). When consider the cost of each step and want to find a path with least cost, it is straightforward to adapt BFS to UCS, which uses a priority queue to choose and pop the frontier with least cost. When encounter node which is already in frontier, we have to update the frontier if current node has less cost. By doing this, we can find optimal solution. Here we implement UCS search results are shown below.

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

D. A* search

UCS algorithm depends on the cost from start node to current node, however, the cost from current node to goal is still extremely useful. Calculating accurate cost is hard and time-consuming, so we use heuristic to estimate it. The priority queue of frontiers sort items according to the evaluation function: f(k)=g(k)+h(k), where g(k) is the cost of the path from start node to the node k and h(k) heuristic estimate of the cost of achieving the goal from k. To guarantee the completeness and optimality in graph search, we have h(A)-h(B)<=cost(A-B), although we have to apply this principle for all states, actually, just adjacent states are OK.

By considering problem relaxation, we can easily design heuristic function hand-crafted. Satisfy optimality is a tricky part. In A* search, we just use manhattanHeuristic to test. Comparison between A* and UCS in bigMaze is shown below.

Layout	Total Cost	Nodes expanded	Score
A*	210	549	300
UCS	210	620	300

To find difference between various search algorithms, I use openMaze to test.

Algorithm	Total Cost	Nodes expanded
DFS	298	576
BFS	54	679
UCS	54	682
A*	54	535

A* Search significantly reduce the expanded nodes.

3. Corners Problem

A. Finding All the Corners

In this question, we should change SearchAgent for the task about eating four dots at each corner of the maze. One major problem is using abstract state representation to show whether all four corners have been reached. I use a nested tuple like (current_position, (unvisited corners)). At the beginning, I use list to store information, however, it is unhashable, which causes me lot trouble.

Here we implement Corners Problem search results are shown below.

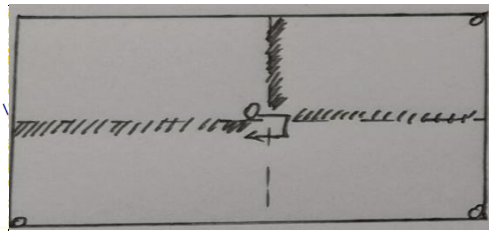
Maze	Total Cost	Nodes expanded	Score
tinyCorners	28	243	512
mediumCorners	106	1921	434
bigCorners	162	7862	378

B. Corners Problem: Heuristic

As A* search above, we use manhattanHeuristic to relax problem. For unvisited corners, corner with the largest manhattan distance is the last one to visit. Once upon reaching the corner, we finish. So my first Heuristics function is using maximum manhattan distance. It's obvious consistent because the maximum manhattan distance will not reduce over 1.

Consider that if we have already reached a corner, we should walk along the short side to get the second corner, then the long side to get the third corner, then the short side to get the last corner. The code is shown below. However, when three corners left, there is a special situation breaking the consistence. I draw this situation and show it.

```
unvisit = list(state[1])
result = 0
state_iter = state[0]
while unvisit:
    distance, corner = min([(util.manhattanDistance(state_iter, corner), corner) \
                            for corner in unvisit])
    result = result + distance
    state_iter = corner
    unvisit.remove(corner)
return result
```



So when three corners left, we should handle it in another way. I design a adopted function based on original one.

```
unvisit = list(state[1])
result = 0
state_iter = state[0]
#special case
#distance means longside+shortside
if len(unvisit) == 3:
    result = min([util.manhattanDistance(state_iter, corner) \
                  for corner in unvisit]) + problem.distance
    return result
else:
    while unvisit:
        distance, corner = min([(util.manhattanDistance(state_iter, corner), corne
                                 for corner in unvisit])
        result = result + distance
        state_iter = corner
        unvisit.remove(corner)
    return result
```

I can prove when 4->3 and 3->2 is correct. When 4->3, original heuristic=1+2*short side+long side, next heuristic=short side(closest)+short side+long side(distance).
When 3->2, original heuristic= 1 (closest)+short side+long side(distance).
, next heuristic= short side(closest)+long side.
When I test the consistence, I writer codes in A* of search.py. The result approves my design.

```
for successror in successors:
    p_heurisic = heuristic(state_iter,problem)
    child_node = Anode(successror[0], node_iter, node_iter.path_cost + successror[2], successror[1],heuristic(successror[0],problem))
    if p_heurisic > (child_node.heuristic + 1): print("False")
```

Heuristics	Maze	Total Cost	Nodes expanded
1	mediumMaze	106	1136
2	mediumMaze	106	774

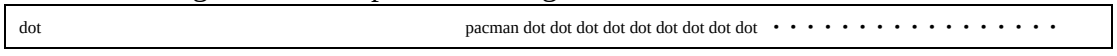
4. Eating all dots

A. Eating All The Dots

It is similar to Corners problem. We can use dots to replace corners, dot with the largest manhattan distance is the last one to visit. Once upon reaching the dot, we finish. But this task is not so simple as Corners problem, because we do not know the dot position. So it is reasonable to replace largest manhattan distance by the largest true path distance in maze. It is admissible because we only calculate the distance between farthest dot and current position. We can use the search algorithm which we write.It will take some time, at the beginning, I do not use heuristicInfo to store information, which causes a lot of repeated calculation. My algorithms runs 30s!!! Then I change to get better answer. I spend 4.4 seconds and expand 4092 nodes to solve trickySearch. Unfortunately, I do not design effective Heuristics function to solve mediumSearch.

B. Suboptimal Search

Use greedy Search to solve problem. Actually this question aims to test to understand and use the class in python. And I use A* search.
I 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.



It obviously that pacman turning left first results shortest path, but it will turn right first because it try to eat the nearest dot.