Artificial Intelligence Project Report: Search in Pac-Man

Zhankui He

15307130175@fudan.edu.cn

***Abstract*— Pac-Man is a classic game aimed at finding short paths in a maze, eating more dots and avoiding ghosts. In this assignment, we implement an intelligent agent to undertake those tasks like a human player. The agent acts based on several specific *Search* algorithms, such as *DFS, BFS, UCS* and *A** with some well-designed *heuristic* functions.**

I. INTRODUCTION

In our memory, Pac-Man is a classic game aimed at finding short paths in a maze. The assignment is based the classic game and we should achieve all these challenges by programming algorithms to design an intelligent agent.

This assignment contains eight questions(Q1-Q8). In this report, they are organized as three sections according to the topics as follows:

- **Q1-Q4:** The task focus on finding a fixed food dot, where several search strategies need to be implemented, and those questions are called as *one dot problem* in Section II.
- **Q5-Q6:** The task requires the agent reaching all corners of maze, where the special agent and heuristic function will be realized. We will concentrate on these questions called as *corners problem* in Section III.
- **Q7-Q8:** This task aims at eating all food dots in as few steps as possible, where we will design a new heuristic function and agent. The questions summarized as *dots problem* will be illustrated in Section IV.

II. ONE DOT PROBLEM

A. Question Analysis

According to the tasks proposed from Q1 to Q4, we know the intelligent agent should focus on finding a fixed food dot in a maze. Therefore, regarding the position of the pac-man as the agent's *state*, the maze can be solved as a search problem of the state space, which is formulated as follows:

- **States:** The state is determined by the agent's position (x, y), where x and y denote the coordinate horizontally and vertically.
- **Initial state:** Agent's start position, set as (x_0, y_0).
- **Actions:** In the maze, the agent at most can go one step *North*, *South*, *East* and *West* if there is no wall blocked in the specific direction.
- **Transition model:** Given a state and action, this returns the resulting state, which is a step movement in one of the legal directions.
- **Goal Test:** Is there a food dot in this agent's current location?
- **Path Cost:** Each step costs 1, so the path is the length of the path to reach the food.

Concentrated on the specific search problems in Q1-Q4, we should realize four specific search strategies called *Depth-First Search, Breadth-First Search, Uniform-cost Search* and *A-star Search*. As we know, these algorithms are based on the same generic Graph Search algorithm but with different updating strategy for their fringes. So we decide to realize the Generic Graph-Search algorithm at first, then we use different data structures to update their own fringes.

B. Generic Graph Search

The original pseudo-code of Graph Search algorithm runs as Algorithm 1:

However, there is an implementation problem. The provided code of *fringe* shows that we can only get the *state* from Pop(*fringe*), otherwise the *Original Graph Search* will go wrong when judging whether a state is in the *fringe*.

Hence, we store *state* in *fringe* and propose a **Dictionary** to map *node* to *state*, which follows the

[1]This work is a project report for chapter *Solving Problems by Searching* in course *Artificial Intelligence*, *Fudan* University. The project is adapted from the *Berkeley Pac-Man Assignments* originally created by *John DeNero* and *Dan Klein*.

[2] Code is in `https://github.com/AaronHeee/`.

Algorithm 1: Original Graph Search

Input: problem
Output: a solution, or failure

```
1  node ← a node with Initial-State,
     Path-Cost=0
2  fringe ← a specific data structure with node
3  explored ← an empty set
4  while do
5      if the fringe is empty then
6          return failure
7      node ← Pop(fringe)
8      if problem.Goal-Test(node.State) then
9          return Solution(node)
10     add node.State to explored
11     for each action in
         problem.Actions(node.State) do
12         child ← Child-Node(problem, node,
             action)
13         if child.State is not in the fringe or
             explored then
14             fringe ← Insert(child, fringe)
```

update rule as the same as *fringe*. So In Algorithm 2: Generic Graph Search, we change *line 7* in Algorithm 1 into that:

node ← Dictionary.lookup(Pop(*fringe*)).

C. *Q1: Depth-First Search*

In Depth-First Search, we implement the *fringe* with **Stack** data structure, where *state* will be maintained as FIFO queue to explore deepest level of the search tree. The algorithm is explained in Section II-B, and the results are as follows:

Type of Maze	Cost	Score	Expended Nodes
Tiny Maze	10	500	15
Medium Maze	130	380	146
Big Maze	210	300	390

Because DFS expends the deepest node in the current fringe, while this node might not be the goal for the agent. Thus, the actual solution need not include all the explored squares on the way to the goal.

D. *Q2: Breadth-First Search*

In Breadth-First Search, we implement the *fringe* with **Queue** data structure, where *state* will be maintained as a FILO queue. The generic search algorithm is explained in Section II-B, and the results of BFS are:

Type of Maze	Cost	Score	Expended Nodes
Medium Maze	68	442	269
Big Maze	210	300	620

Also, because BFS is generic and optimal, it can solve other problems such as *Eight-Puzzle*.

E. *Q3: Uniform-cost Search*

The fringe is a little different in UCS, where we maintain a **Priority Queue**, which return the *state* with the smallest cost to reach the node denoted as $g(n)$ at one time. We use *Minimum Heap* to realize the *Priority Queue* as the *fringe*. And the experiments results:

Type of Maze	Cost	Score	Expended Nodes
Medium Maze	68	442	269
Big Maze	210	300	1240
Dotted Maze	1	300	620
Scary Maze	68719479864	418	216

According to the table, I get very low and very high path costs for the $3rd$ and $4th$ one respectively, due to their exponential cost functions for their agents.

F. *Q4: A* Search*

We also use **Priority Queue** to maintain *fringe* as the same as UCS in Section II-E. The only difference lays on the cost function, which is $g(n) + h(n)$ instead of only $g(n)$ in UCS, where $h(n)$ denotes the heuristics cost from this node to the goal. With *Manhattan Distance* heuristics function used, the experiments results are:

Type of Maze	Cost	Score	Expended Nodes
Big Maze	210	300	549
Open Maze	54	456	535

It's obvious for A* that the number of expended nodes in Big Maze is much less than that via UCS in in Section II-E.

III. CORNERS PROBLEM

A. *Question Analysis*

Compared with tasks from Q1-Q4, the Corner Problem focus on a new goal to reach four corners of a maze. So the *State*, *Initial State* and *Goal Test* can be converted into:

- **States:** A tuple of agent's position (x, y), and untouched corner list $C \in P(\{c_1, c_2, c_3, c_4\})$ where c denotes the corner.
- **Initial state:** Agent's start position and corners of this maze to be visited, set as $((x_0, y_0), \{c_1, c_2, c_3, c_4\})$.
- **Actions:** If agent visit corner c_i, c_i will be deleted from the untouched corner list. And the rest are the same as *Actions* in Section II-A
- **Goal Test:** Is untouched corner list $C = \emptyset$?

B. *Q5: Finding All the Corners*

So the key is to implement the formulated corner problem stated clearly in Section III-A. The performance of BFS in this type of problems are:

Type of Maze	Cost	Score	Expended Nodes
Tiny Corner	28	512	252
Medium Corner	106	434	1966

And it's acknowledged that if the path cost is a nondecreasing function of the depth of the node in search tree, the solution of BFS is optimal. Hence in the problem, as the path-cost defined as Section II-A is definitely nondecreasing, the proposed path is the shortest.

C. *Q6: Heuristic Function of Corner Problem*

*1) **Heuristic 1: The Longest Manhattan Distance**:*

$$H_1((pos, C)) = max\{MD(pos, c_i)|c_i \in C\}$$

where pos denotes the current position, MD denotes the Manhattan Distance. We know that Manhattan Distance is the shortest path cost when there is no wall. So the heuristic function is admissible. And it's easy to know that for every action with cost z, there is a drop in heuristic is not more than z. Otherwise, z, instead of Manhattan Distance, is the cost of shortest path, which is contradictory obviously. So this heuristic function is consistent.

*2) **Heuristic 2: The Enhanced Longest Manhattan Distance**:*

$$H_2((pos, C)) = min\{MD(pos, c_i)+MD_{max}(c_i, C)\}$$

$$MD_{max}(c_i, C) = max\{MD(c_i, c_j)|c_j \in C/\{c_i\}\}$$

These above are in the condition that the elements in C is not less than 2, otherwise it's the same as $Heuristic_1$. And we know that when $|C| \leq 2$, $min\{MD(pos, c_i) + MD_{max}(c_i, C)\}$ is the shortest path when there is no wall, and compared with H_1, it approximates the real situation better. And that is admissible and consistent as what illustrated in Section III-C.1.

The performances of these experiments run as below:

Type of Maze	Heuristics	Cost	Expended Nodes
Medium Corner	H_1	106	1136
Medium Corner	H_2	106	834
Big Corner	H_1	162	4380
Big Corner	H_2	162	3118

IV. DOTS PROBLEM

A. *Question Analysis*

This tasks are all about eating dots. So the *State*, *Initial State* and *Goal Test* can be converted about foods' positions:

- **States:** A tuple of agent's position (x, y), and food Grid denoted as $foods$.
- **Initial state:** Agent's start position and all of the foods' sites in this maze to be visited, set as $((x_0, y_0), Foods)$.
- **Actions:** If agent visit position with a dot, the food Grid $foods$ will be updated. And the rest are the same as *Actions* in Section II-A
- **Goal Test:** Is the list from food Grid empty?

B. *Q7: Eating All The Dots*

*1) **Heuristic 1: Longest Manhattan Distance**:* With exploration experience in Section III-C, I derive the heuristic function H_1 as the same as Section III-C.1.

$$H_1((pos, foods)) = max\{MD(pos, f)|f \in foods\}$$

The consistency has been explained in Section III-C.1. Unfortunately, the performance is poor as the expended node in *tricky search* is 9551. So more efficient heuristic function should be derived.

*2) **Heuristic 2: Longest BFS Distance**:* The idea comes into my mind is to use heuristic function which is more closest to the real path cost. So we can take the *walls* information into consideration. The way to utilize the *walls* of maze is replacing the Manhattan Distance with the BFS path length. As we know, the distance from BFS is the optimal distance between two position.

$$H_2((pos, foods)) = max\{BD(pos, f)|f \in foods\}$$

where BD denotes the distance of two position from BFS. The admissibility and consistency can be discovered like Section III-C.1 and the function is better than H_1 with 4137 nodes expended in *trickySearch*.

*3) **Heuristic 3: The Enhanced Longest BFS Distance**:* The heuristic function is derived naturally like idea of Section III-C.2: if there are more than two dots, the perfect distance must be the minimal one of *sum of postion-to-food distance and the longest food-to-food distance.*

$$H_3((pos, F)) = min\{BD(pos, f_i)+BD_{max}(f_i, F)\}$$

$$BD_{max}(f_i, F) = max\{BD(f_i, f_j)|f_j \in F/\{f_i\}\}$$

where F is $foods$ for short. And the function is definitely better than H_2 with expended node 2230.

The experiments about *trickySearch* run as follows :

Heuristics	Time	Cost	Expended Nodes
H_1	**11.7**	60	9551
H_2	**3.9**	60	4137
H_3	**2.0**	60	2230

Besides, I try some ways to solve the medium search, which expend much less nodes and run faster. However, as you may know, they fails the admissibility or consistency tests.

*4) **Inconsistent Heuristic 1: The Longest Sampling BFS Distance**:* From my perspective, the state space of *mediumSearch* is too large to explore within 60 seconds. So the idea about *sampling* comes into my mind, but that is obvious that the heuristic depends on randomly sampling, so consistency is not guaranteed.

*5) **Inconsistent Heuristic 2: The Greedy BFS Distance**:* I set the heuristic function as the sum of position-to-food distance and one food to its closest food recursively.To speed up, the $problem.heuristicInfo$ is used to memorize the path length. The algorithm is fast extremely, however, it's not admissible because the greedy cost may not the optimal one.

Still, I share the results of these heuristic functions. Although they are not admissible or consistent, they are some explorations anyway.

Heuristics	Maze	Time	Cost	Expended Nodes
IH_1	Tricky	0.2	60	88
IH_2	Tricky	0.3	60	91
IH_1	Medium	8.5	166	664
IH_2	Medium	5.6	155	216

C. *Q8: Suboptimal Search*

For this task, we should construct a kind of greedy search algorithm which reach the closest food dot each time. The problem defined in $AnyFoodSearchProblem$ sets the *state* as the position. And the *Goal-test* is if the agent reach the position of the closest dot. As the problem is already defined for us according to the code in function $findPathToClosestDot()$, so it is convenient for us to implement the algorithm by solving the $AnyFoodSearchProblem$ with the search algorithm such as *BFS*, *UCS* and *Astar*.

And the results are as below:

Maze	Cost	Score
Medium	171	1409
Big	350	2360

It can solve the maze, but as we know, in this problem the local optimal solution doesn't have to be the global optimal one. So the search solution is suboptimal.