Report of Project 1 of the Course Artificial Intelligence: Search in Pac-Man

20300180133 Xinyin Liu , 20300680217 Jingwen Li

October 10, 2023

ABSTRACT

Pac-Man is an iconic game that revolves around navigating mazes, gobbling up dots, and evading ghosts. In this assignment, we have developed an intelligent agent to perform these tasks in a manner similar to a human player. The agent's actions are guided by various search algorithms, including DFS, BFS, UCS, and A*, along with heuristic functions.

1 INTRODUCTION

Pac-Man is a classic timeless game centered around the pursuit of optimal pathways within intricate mazes. This assignment draws its inspiration from this classic game, requiring us to tackle a series of challenges through algorithmic programming in order to create an intelligent agent.

This assignment encompasses a total of eight questions (Q1-Q8). In this report, we have structured them into three distinct sections based on their respective themes:

Questions Q1-Q4 focus on the task of locating a single fixed food dot. These questions involve the implementation of various search strategies and are collectively referred to as the "One Dot Problem" in Section II.

Questions Q5-Q6 revolve around the objective of guiding the agent to reach all corners of the maze. To achieve this, we will explore the utilization of a special agent and heuristic functions. These questions are categorized as the "Corners Problem" in Section III.

Questions Q7-Q8 are centered on the challenge of consuming all food dots in the fewest possible moves. In pursuit of this goal, we will design a novel heuristic function and agent. These questions are grouped under the "Dots Problem" and will be discussed in detail in Section IV.

2 ONE DOT PROBLEM

2.1 Analysis of Questions

Given the tasks outlined in questions Q1 to Q4, it is clear that the primary objective of the intelligent agent is to locate a fixed food dot within a maze. Consequently, the maze-solving problem can be treated as a search problem within a state space. This problem can be formulated as follows:

1. States: The state is defined by the agent's position (x, y), where x and y represent horizontal and vertical coordinates.
2. Initial state: The agent's starting position, denoted as (x0, y0).
3. Actions: Within the maze, the agent can move one step North, South, East, or West, provided there are no obstacles blocking the chosen direction.
4. Transition model: Given a state and an action, this model yields the resulting state, which represents a single step movement in one of the legal directions.
5. Goal Test: Is there a food dot present at the agent's current location?
6. Path Cost: Each step incurs a cost of 1, making the path length equal to the number of steps required to reach the food.

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

Fig. 1. Graph Search pseudo-code

In focusing on the specific search problems presented in Q1-Q4, we aim to implement four distinct search strategies: Depth-First Search, Breadth-First Search, Uniform-Cost Search, and A-Star Search. These algorithms are all derived from the same generic Graph Search algorithm but employ different strategies for updating their fringes. Therefore, we begin by implementing the Generic Graph-Search algorithm and subsequently use various data structures to manage their respective fringes.

2.2 Graph Search pseudo-code

The original pseudo-code for the Graph Search algorithm is provided in Figure 1:

2.3 Q1: Depth-First Search

In the Depth-First Search (DFS) approach, we utilize a Stack data structure to manage the fringe, where states are maintained as a First-In-First-Out (FIFO) queue to explore the deepest levels of the search tree first. The findings are summarized below:

Table 1. Summary of DFS Results

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

DFS explores the deepest node within the current fringe, which may not necessarily be the agent's ultimate goal. Consequently, the actual solution may not include all the explored squares encountered on the path to the goal.

2.4 Q2: Breadth-First Search

In the context of Breadth-First Search (BFS), we employ a Queue data structure to manage the fringe, where states are maintained as a First-In-Last-Out (FILO) queue. The comprehensive details of the generic search algorithm can be found above, and the outcomes obtained through BFS are summarized below:

Table 2. Summary of BFS Results

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

Additionally, BFS exhibits generality and optimality, making it suitable for solving various problems, including but not limited to the Eight-Puzzle.

2.5 Q3: Uniform-Cost Search

In Uniform-Cost Search (UCS), the fringe operates differently, as we maintain a Priority Queue that returns the state with the smallest cost to reach the node, denoted as g(n), at each iteration. To implement the Priority Queue as the fringe, we employ a Minimum Heap data structure. Here are the experimental results:

Table 3. Results of Uniform-Cost Search (UCS)

Maze Type	Cost	Score	Expanded Nodes
Medium Maze	68	442	269
Big Maze	210	300	1240
Medium Dotted Maze	1	646	186
Medium ScaryMaze	68719479864	418	108

As indicated by the table, the 3rd and 4th mazes exhibit notably low and exceptionally high path costs, respectively. This is primarily due to the exponential cost functions employed for their agents.

2.6 Q4: A* Search

In A* Search, we also employ a Priority Queue to manage the fringe, following the same approach as in UCS. The primary distinction lies in the cost function, which combines g(n) and h(n), unlike UCS, where only g(n) is considered. Here, h(n) represents the heuristic cost from the current node to the goal. Utilizing the Manhattan Distance heuristic function, the experimental results are as follows:

Table 4. Results of A* Search

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

It's evident that in the case of A*, the number of expanded nodes in the Big Maze is significantly lower compared to that of UCS, as discussed above.

3 CORNERS PROBLEM

3.1 Analyzing the Problem

In contrast to the tasks presented in Q1-Q4, the Corners Problem introduces a new objective: reaching all four corners of a maze. Consequently, we can redefine the State, Initial State, and Goal Test as follows:

1. States: A tuple comprising the agent's position (x, y) and an untouched corner list, denoted as $c \in \{c_1, c_2, c_3, c_4\}$, where 'c' represents a corner.
2. Initial state: The agent's initial position and the list of corners in the maze that need to be visited are set as $((x_0, y_0), \{c_1, c_2, c_3, c_4\})$.
3. Actions: If the agent visits corner c_i, c_i will be removed from the untouched corner list. The remaining actions are consistent with those outlined in Section 2.
4. Goal Test: Is the untouched corner list $C = \emptyset$?
5. Path Cost: Each step incurs a cost of 1, making the path length equal to the number of steps required to reach the food.

3.2 Q5: Locating All Four Corners

The critical aspect here is to effectively implement the formulated corner problem, as outlined clearly in Section III-A. The performance of Breadth-First Search (BFS) in such scenarios is as follows:

Table 5. Performance of BFS in Locating All Four Corners

Maze Type	Path Cost	Score	Expanded Nodes
Tiny Corner	28	512	435
Medium Corner	106	434	2448

It's well-recognized that when the path cost is a non-decreasing function of the depth of the node in the search tree, BFS provides an optimal solution. Therefore, in this problem, as the path

cost, defined in Section 2, is undoubtedly non-decreasing, the proposed path is indeed the shortest possible.

3.3 Q6: Heuristic Function of Corner Problem

Heuristic function: General Manhattan Distance (with repetitive lines deleted) We use $height(A,B)$ and $breadth(A,B)$ to denote the y-distance and the x-distance between two coordinates respectively. Then we have the following piecewise Heuristic function:

$$H(x) = \begin{cases} length + 2 \times width & \text{if len is 4} \\ \sum_{i=1}^{3} height(pos, food(i)) & \text{if len is 3} \\ length + width & \text{if len is 2 and corners are diagonal} \\ length + height(pos, food(1)) & \text{if len is 2 and corners on the top or bottom side} \\ width + breadth(pos, food(1)) & \text{if len is 2 and corners on the left or right side} \end{cases}$$

Note that if $width > length$, we tilt the two coordinates in the code beforehand. Thus, we can assume $length > width$.

We only consider a one-grid move, because general moves can be pieced together step by step. Since the change in H for each move does not exceed 1 and the cost for each move is always 1, the consistency is proven.

Table 6. Performance of BFS in Locating All Four Corners

Maze Type	Path Cost	Score	Expanded Nodes
Medium Corner	106	434	843
Big Corner	162	378	3073

4 DOTS PROBLEM

4.1 Analyzing the Problem

All of these tasks revolve around the objective of consuming dots. Therefore, the State, Initial State, and Goal Test can be reformulated in terms of the positions of the foods as follows:

1. States: A tuple representing the agent's position (x, y) and the food grid, denoted as "foods."
2. Initial state: The agent's initial position and all the locations of the food in this maze that need to be visited, set as $((x_0, y_0), Foods)$.
3. Actions: When the agent visits a position with a dot, the food grid "foods" will be updated accordingly. The remaining actions align with those detailed discussed above.
4. Goal Test: Is the list of foods in the food grid empty?
5. Path Cost: Each step incurs a cost of 1, making the path length equal to the number of steps required to reach the food.

4.2 Q7: Eating All The Dots

5 Q7

Heuristic function: MinMax Manhattan Distance The function $H(x)$ is defined as:

$$H(x) = \begin{cases} \min\limits_{1 \le i \le n} (pos, food(i)) + \max\limits_{1 \le i < j \le n} (food(i), food(j)) & \text{if } n \ge 2 \\ \min(pos, food) & \text{if } n = 1 \\ 0 & \text{otherwise} \end{cases}$$

We still only consider a one-step move. If this step does not consume food0, then $\max(\text{food}(i), \text{food}(j))$ remains unchanged, and the change in $\min(\text{pos}, \text{food}(i)) \le 1$. In this case, it's consistent. If food0 is consumed which means Pacman is just one step closer to food0, we assume the subsequent $\min(\text{pos}, \text{food})$ corresponds to food(t) which is the closest food to Pacman, then there are three scenarios:

1. If food0 is unrelated to $\max(\text{food}(i), \text{food}(j))$, then the difference in the heuristic function i.e h(0)-h(t) ≤ 0, so it's definitely smaller than the cost which equals to 1.
2. If food0 is related to $\max(\text{food}(i), \text{food}(j))$ and food(t) is the subsequent argmax, then given that we're considering the Manhattan distance, simple geometry tells us that $h(0) - h(t) = \text{cost}$.
3. If food0 is related to $\max(\text{food}(i), \text{food}(j))$ and food(t) is not the subsequent argmax, then the max value in $h(t)$ will be greater than case 2, so $h(0) - h(t) \le \text{cost}$.

Table 7. Performance of BFS in Locating All Four Corners

Maze Type	Path Cost	Score	Expanded Nodes
TrickySearch	60	570	7798

5.1 Q8: Suboptimal Search

For this particular task, we are tasked with devising a type of greedy search algorithm that consistently targets the closest food dot at each step. The problem, as defined in the Any Food Search Problem, represents the state as a position. The goal test checks if the agent has reached the position of the closest dot. Given that the problem's definition is already provided for us through the code in the function findPathToClosestDot(), it is convenient for us to implement the algorithm by solving the AnyFoodSearchProblem using search algorithms such as BFS, UCS, and A*. The resulting outcomes are as follows:

Table 8. Results of Suboptimal Search

Maze	Cost	Score
Medium	171	1409
Big	350	2360

While this approach effectively solves the maze, it's worth noting that in this problem, the locally optimal solution doesn't necessarily guarantee global optimality. Therefore, the search solution is suboptimal.