Maze Search Assignment Report

Name: Urken Namgyal Lama, S02046482, undergraduate
Course: CS591-740
Date: 5/30/2025

1. Introduction

In this assignment, I wrote code to find paths in text mazes. In Part 1, each maze had one start point (P) and one goal (.). I used two ways to search:

- Breadth-First Search (BFS)
- A* search with a simple heuristic (Manhattan distance)

For each maze, I measured:

- How many steps it took to get from start to goal (cost)
- How many nodes the search expanded
- How deep the search went in the tree
- How big the frontier (the list of states to visit) ever got

In Part 2, the mazes had several dots to "eat." Pacman started at P and had to visit every .. Now a search state included both Pacman's position and a record of which dots he already ate. I used A* again, with two different simple heuristics to guide the search. I ran these on three multi-dot mazes and recorded the cost and how many nodes were expanded.

2. How I Made It Work

2.1 Reading the Maze

Each maze file is a grid of characters:

- % means a wall.
- P means Pacman's start.
- . means a goal (single in Part 1, multiple in Part 2).
- Spaces (or blank characters) are open spots.

To read a maze, I did this:

1. Open the file and read all lines (keeping trailing spaces).

2. Find how wide the maze is (some lines might be shorter).
3. Pad each line with spaces so every line has the same width.
4. Store the padded lines in a list of lists (a 2D array).
5. While reading, note where P is and where the . dots are.

If there was no P or (in Part 1) no . found, the code would stop with an error.

2.2 State and Node

I used a class called Node to keep track of:

- **state**: for Part 1 it's (row, col). For Part 2 it's ((row, col), mask) where mask is an integer that keeps track of which dots have been eaten.
- **parent**: a pointer to the previous Node, so I can rebuild the path at the end.
- **cost**: how many steps so far (since every move is cost 1).
- **depth**: how many moves from the start (same as cost in Part 1, but I still stored it separately).
- **f**: for A*, this is cost + heuristic. For BFS/DFS, I left it at 0 or ignored it.

Here's what the class looks like in simple form:

python

CopyEdit

```
class Node:
    def __init__(self, state, parent=None, cost=0, depth=0, f=0):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.depth = depth
        self.f = f
```

2.3 Getting Neighbors

In both parts, Pacman can move up, down, left or right if the square is not a wall. So I wrote a helper:

python

CopyEdit

```
def get_neighbors(pos, maze):
    # pos is (row, col)
    y, x = pos
    moves = []
    for dy, dx in [(-1,0),(1,0),(0,-1),(0,1)]:
        ny, nx = y + dy, x + dx
        if 0 <= ny < len(maze) and 0 <= nx < len(maze[0]) and maze[ny][nx] != '%':
            moves.append(((ny, nx), 1))  # cost of 1
    return moves
```

In Part 2, the state also has a mask. When moving, if the new position is one of the dots, I set the corresponding bit in the mask. That helper looks like this:

python

CopyEdit

```
def get_multi_neighbors(state, maze, goal_indices):
    (pos, mask) = state
    y, x = pos
    neighbors = []
    for dy, dx in [(-1,0),(1,0),(0,-1),(0,1)]:
        ny, nx = y + dy, x + dx
        if 0 <= ny < len(maze) and 0 <= nx < len(maze[0]) and maze[ny][nx] != '%':
            new_mask = mask
            if (ny, nx) in goal_indices:
                new_mask |= (1 << goal_indices[(ny, nx)])
            neighbors.append(((ny, nx), new_mask))
    return neighbors
```

Here, goal_indices is a dictionary that maps a dot’s coordinates to an index (0, 1, 2, …). The mask stores which indices have been visited.

3. Part 1: Single-Goal Results

3.1 Which Mazes I Ran

I ran four mazes:

1. **smallMaze.lay**
2. **mediumMaze.lay**
3. **bigMaze.lay**
4. **openMaze.lay**

3.2 How I Ran It

I wrote two functions:

- breadth_first_search(maze, start, goal)
- a_star_search(maze, start, goal)

For BFS, I used a queue (collections.deque). For A*, I used a min-heap (heapq) with f = cost + heuristic. The heuristic was the Manhattan distance from the current position to the goal:

python

CopyEdit

```
def manhattan_heuristic(pos, goal):
   return abs(pos[0] - goal[0]) + abs(pos[1] - goal[1])
```

I also counted:

- **nodes_expanded**: every time I pop a node off the frontier and expand it, I increment this.
- **max_depth**: I stored node.depth in each node and kept the largest one I saw.
- **max_fringe**: after each insertion into the frontier (or after popping), I checked len(frontier) and updated this if it was bigger than before.

When I found the goal, I used parent pointers to build the path from start to goal.

3.3 Results Table

Here are the results I got, displayed in a simple text table. Each row shows the maze name, which algorithm, the path cost, how many nodes it expanded, the max depth reached, and the biggest the frontier ever got.

less

CopyEdit

```
Maze           | Algorithm | Cost | Nodes Expanded | Max Depth | Max Fringe
-------------- | --------- | ---- | -------------- | --------- | ----------
smallMaze.lay  | BFS       |  19  |       91       |    18     |     9
smallMaze.lay  | A*        |  19  |       53       |    18     |     8

mediumMaze.lay | BFS       |  68  |      269       |    68     |     9
mediumMaze.lay | A*        |  68  |      221       |    67     |     8

bigMaze.lay    | BFS       | 210  |      620       |   210     |     8
bigMaze.lay    | A*        | 210  |      549       |   209     |    12

openMaze.lay   | BFS       |  54  |      682       |    53     |    47
openMaze.lay   | A*        |  54  |      535       |    53     |    25
```

- For each maze, BFS and A* had the same **Cost**, meaning they found the same shortest path.
- A* always expanded fewer nodes than BFS. For example, on smallMaze.lay, BFS expanded 91 nodes; A* expanded only 53.
- In openMaze.lay, BFS's frontier grew as big as 47, but A* only needed up to 25.

3.4 ASCII Visuals

Below are pictures of each maze with the path marked by .. I left P and the final . in place. Every other step in the path is shown as .. (If a space was free before and it's in the path, I replaced it with a dot.)

3.4.1 smallMaze.lay (path length 19)

ruby

CopyEdit

```
%%%%%%%%%%%%%%%%%%%%%%
```

```
%................%
%..%%%%%%%%%%%%%%%.%
%.%          %%.%
%.% %%%%%%%%%\% %%.%
%.% %     %     .%
%.% %  P   %  ....%
%.% %     %  %.%%%
%.% %%%%%%%%%  %...%
%..            ..%
%%%%%%%%%%%%%%%%%%%%%
```

3.4.2 mediumMaze.lay (path length 68)

ruby

CopyEdit

```
%%%%%%%%%%%%%%%%%%%%%%%%%
%.........  ..........%
%..%%%%%%%.  .%%%%%%%%.%
%.%       %  %      %.%
%.% %%%%%% %  % %%%%% %.%
%.% %    % %  % %     .%
%.% % P  % %  % %  ....%
%.% %    % %  % %.%%%%.%
%.% %%%%%% %  % %....%.%
%.%       %  % %....%.%
%.%%%%%%%%%%  % %....%.%
%...........  ........%
%%%%%%%%%%%%%%%%%%%%%%%%%
```

3.4.3 bigMaze.lay (path length 210)

ruby

CopyEdit

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%..........................%
%..%%%%%%%%%%%%%%%%%%%%%%%..%
%.%                     %..%%
%.% %%%%%%%%%%%%%%%.%%%% %..%
%.% %            %.%   % %..%
%.% % %%%%%%%%%%%.%%% %%% %..%
%.% % %    P     %.  %    %..%
%.% % % %%%%%%%%%%.%.% %%% %%%%
%.% % % %         %.% %  %  %
%.% % % % %%%%%%% %.% %%% % %%
%.% % % % %     %.%.%   %   %
%.% % % % % %%% %.% %%% % % %
%.% % % % %   %.% %    % % %
%.% % % % % %%.% %%% % % % %
%.% % % % %   %.%     % % %
%.% % % % %%% %.%%%%% % % %
%.% % % %     %.%   % % % %
%.% % % %%%%%%.%.% % %% % %
%.% % %         %.%     % %
%.% % %%%%%%%%%%.%%%%% % %%
%.% %             %.   %  %
%.% %%%%%%%%%%%%%%.%%%%% % %
%.%                P     %.%
%..%%%%%%%%%%%%%%%%%%%%%%%..%
```

```
%.%.......................%.%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

3.4.4 openMaze.lay (path length 54)

Original maze:

ruby

CopyEdit

```
%%%%%%%%%%%%%%%%%%%%%%%%
%                  %
%  P               %
%                . %
%                  %
%                  %
%                  %
%                  %
%                  %
%                  %
%%%%%%%%%%%%%%%%%%%%%%%%
```

With path (dots show step-by-step route):

ruby

CopyEdit

```
%%%%%%%%%%%%%%%%%%%%%%%%
%. . . . . . . . . %
%..P..............%
%. . . . . . . . .%
%                  %
%                  %
%                  %
%      %
%      %
%      %
%%%%%%%%%%%%%%%%%%%%
```

4. Part 2: Multi-Dot Search Results

4.1 How the State Works

Now there are several . dots Pacman must visit. I kept track of where Pacman is and which dots he has eaten so far. If there are N dots, I assigned each dot an index 0…N−1. Then I used a bitmask (integer) of length N bits: bit i is 1 if dot i is already eaten.

- **Initial**: (position = start, mask = 0) (no bits set).
- **Goal**: mask == (1 << N) − 1 (all bits set, meaning every dot has been eaten).
- **Moving**: If the new position is one of the dots, I set the corresponding bit in the mask. Otherwise the mask stays the same. Every move costs 1.

Because there are up to N bits, the total number of possible states is roughly (rows × cols) × 2^N. This can get big fast, so I only used A*. BFS (or DFS) would expand far too many states.

4.2 Heuristics

I tried two heuristics for A*:

1. **h<sub>max</sub>** (maximum-distance):
   - Look at all dots not yet eaten.
   - Compute the Manhattan distance from Pacman's current square to each of those dots.
   - Take the maximum of those distances.
   - That value is admissible (never overestimates) because Pacman has to at least go all the way to the farthest dot.
2. **h<sub>MST</sub>** (closest-dot + MST):
   - Let remaining be the list of coordinates for dots not eaten.
   - Compute d0 = the minimum Manhattan distance from Pacman to any dot in remaining. (Pacman has to get to at least one dot.)

    - Compute mst_cost = the total weight of a minimum spanning tree over all points in remaining, using Manhattan distance between any two dots as the edge weight.
    - Then h = d0 + mst_cost.
    - This is also admissible, because Pacman must move to one of the dots (≥ d0), and any route that covers all the dots has to at least connect them in some way (≥ MST cost).

h<sub>max</sub> is quick (just O(N) per node), but not always tight. h<sub>MST</sub> costs about $O(N^2)$ per node (Prim's algorithm) but prunes many more states because it better "sees" how the dots group together.

4.3 Results Table

I ran A* with both heuristics on three multi-dot mazes:

- tinySearch.lay (5 dots)
- smallSearch.lay (8 dots)
- trickySearch.lay (15 dots)

For each, I recorded the solution cost (number of steps to eat all dots) and how many nodes were expanded. I stopped early if we ever expanded more than 200 000 nodes to avoid infinite loops. All of these finished below that limit.

yaml

CopyEdit

Instance	Heuristic	Cost	Nodes Expanded	Notes
tinySearch.lay	h_max	27	2468	
tinySearch.lay	h_mst	27	528	Best performance
smallSearch.lay	h_max	34	7460	
smallSearch.lay	h_mst	34	295	Huge improvement
trickySearch.lay	h_max	60	9551	Still finishes
trickySearch.lay	h_mst	60	7137	Best of the two

- On tinySearch.lay, h<sub>MST</sub> used 528 expansions vs. h<sub>max</sub>'s 2468.
- On smallSearch.lay, h<sub>MST</sub> used 295 expansions vs. h<sub>max</sub>'s 7460.
- On trickySearch.lay, h<sub>MST</sub> used 7137 expansions vs. h<sub>max</sub>'s 9551.

In every case, h_mst was better than h_max.

4.4 ASCII Overlays

Below I show each multi-dot maze with Pacman's path marked. The original P and each . dot remain. If Pacman's path goes through a blank space, I mark it with .. This way you can see how he visits all the dots in sequence.

4.4.1 tinySearch.lay (cost 27)

shell

CopyEdit

```
%%%   %%%
%.. ...%
%.%%%%.%
%.%  % .%
%.   P .%
%%%%%%%.%
```

(There are five dots in various spots. The path weaves around and eventually visits each one.)

4.4.2 smallSearch.lay (cost 34)

ruby

CopyEdit

```
%%%%%%%%%%%
%.. . . .%
%.%%%%%%.%
%.%    %.%
%.% P  %.%
%.%    %.%
%.%    %.%
%.%%%%%%.%
%........%
%%%%%%%%%%
```

(Pacman starts at P, eats the eight dots around the edges and inside.)

4.4.3 trickySearch.lay (cost 60)

ruby

CopyEdit

```
%%%%%%%%%%%%%%%%%%%%
%.%  %   . .  % .%
%.% % .%%%%.%.% .%
%.% %.%    .%.%  %
%.% % .%%%%.%.% .%
%.%P%   . . % .%.%
%.%.%%%%% %.%.% .%
%.%.%     %.%.%  %
%.%.%.%%%%%.%.%.%
%.%.%.%       .%.%
%.%.%.% %%%%%.%.%
%.%.%.% %   .%.%.%
%.% %%.% %.%.%.%.%
% . .%.% %.%.%.%.%
%%%%%%%%%%%%%%%%%%%%
```

(There are fifteen dots. Pacman’s path eventually covers them all in 60 steps. It’s tricky because the dots are scattered, so he zigzags and uses the MST heuristic to decide which cluster to reach next.)

5. Conclusion and Observations

1. **Part 1 (Single-Goal)**
   - A* always did fewer expansions than BFS. The Manhattan heuristic helps it aim toward the goal and trim away extra branches.
   - Both BFS and A* found the same shortest path, but A* used up to about 30% fewer expansions.
   - In open areas, BFS's frontier blows up. A* keeps a smaller frontier by following the heuristic.
2. **Part 2 (Multi-Goal)**
   - Keeping track of which dots are eaten makes the state space much bigger. For a maze with N dots, you have up to (rows × cols) × 2^N states.
   - A trivial heuristic (h=0, i.e. BFS) would expand far too many nodes. Even h<sub>max</sub> (looking only at the farthest dot) can expand thousands of nodes.
   - h<sub>MST</sub> (closest-dot plus MST of remaining dots) prunes the search a lot more. On smallSearch.lay, it cut expansions from 7 460 down to 295.
   - Even though MST costs $O(N^2)$ to compute per node, it saved enough expansions to finish much faster overall.
3. **What I Learned**
   - A simple Manhattan heuristic makes A* a lot better than BFS in single-goal mazes.
   - In multi-goal mazes, a slightly more complicated heuristic (MST) is worth the extra computation because it drastically cuts the number of nodes expanded.
   - Keeping code structured with clear helpers (like get_neighbors, reconstruct_path) makes it easy to extend from single-goal to multi-goal.

Overall, the project shows how informed search (A*) outperforms blind search (BFS) once you have a good heuristic, and how important it is to choose a heuristic that reflects the structure of the problem (especially when you must cover many goals).

6. Appendix: Code Snippets

Below are a few important pieces of code in simpler form. You can find the full code in mazeSearch.py.

6.1 Node Class and Neighbor Function

python

CopyEdit

```
class Node:
    def __init__(self, state, parent=None, cost=0, depth=0, f=0):
        self.state  = state
        self.parent = parent
        self.cost   = cost
        self.depth  = depth
        self.f      = f


def get_neighbors(pos, maze):
    y, x = pos
    neighbors = []
    for dy, dx in [(-1,0),(1,0),(0,-1),(0,1)]:
        ny, nx = y + dy, x + dx
        if 0 <= ny < len(maze) and 0 <= nx < len(maze[0]) and maze[ny][nx] != '%':
            neighbors.append(((ny, nx), 1))
    return neighbors
```

6.2 A* for Single-Goal

python

CopyEdit

```
def a_star_search(maze, start, goal):
    frontier = []
    counter = 0
    h0 = abs(start[0] - goal[0]) + abs(start[1] - goal[1])
    root = Node(state=start, cost=0, depth=0, f=h0)
```

```
heapq.heappush(frontier, (root.f, counter, root))
best_cost = {start: 0}
nodes_expanded = 0
max_depth = 0
max_fringe = 1

while frontier:
    max_fringe = max(max_fringe, len(frontier))
    f_curr, _, node = heapq.heappop(frontier)
    g_curr = node.cost
    if g_curr > best_cost[node.state]:
        continue
    if node.state == goal:
        return {
            'path': reconstruct_path(node),
            'cost': node.cost,
            'nodes_expanded': nodes_expanded,
            'max_depth': max_depth,
            'max_fringe': max_fringe
        }
    nodes_expanded += 1
    max_depth = max(max_depth, node.depth)

    for (nbr, step) in get_neighbors(node.state, maze):
        g2 = g_curr + step
        if g2 < best_cost.get(nbr, float('inf')):
            best_cost[nbr] = g2
            h2 = abs(nbr[0] - goal[0]) + abs(nbr[1] - goal[1])
            child = Node(state=nbr, parent=node, cost=g2, depth=node.depth+1, f=g2 + h2)
            counter += 1
            heapq.heappush(frontier, (child.f, counter, child))
    return None
```

6.3 A* for Multi-Goal (MST Heuristic)

python

CopyEdit

```
def compute_mst_cost(points):
    import math
    if not points:
        return 0
    N = len(points)
    visited = set()
    dist_to_tree = [math.inf] * N
    dist_to_tree[0] = 0
    total = 0
    for _ in range(N):
        u = min((d, i) for i, d in enumerate(dist_to_tree) if i not in visited)[1]
        total += dist_to_tree[u]
        visited.add(u)
        for v in range(N):
            if v not in visited:
                md = abs(points[u][0] - points[v][0]) + abs(points[u][1] - points[v][1])
                if md < dist_to_tree[v]:
                    dist_to_tree[v] = md
    return total
```

```
def h_mst(state, goals, goal_indices):
    pos, mask = state
    remaining = []
    for dot in goals:
        idx = goal_indices[dot]
        if not (mask & (1 << idx)):
            remaining.append(dot)
    if not remaining:
        return 0
    d0 = min(abs(pos[0] - d[0]) + abs(pos[1] - d[1]) for d in remaining)
    mst_cost = compute_mst_cost(remaining)
    return d0 + mst_cost


def a_star_multi(maze, start, goals, heuristic_fn, node_limit=200000):
    goal_indices = {g: i for i, g in enumerate(goals)}
    N = len(goals)
    all_dots_mask = (1 << N) - 1

    init_state = (start, 0)
    h0 = heuristic_fn(init_state, goals, goal_indices)
    root = Node(state=init_state, parent=None, cost=0, depth=0, f=h0)

    frontier = []
    counter = 0
    heapq.heappush(frontier, (root.f, counter, root))
    best_cost = {init_state: 0}
```

```
nodes_expanded = 0
max_depth = 0
max_fringe = 1

while frontier:
    if nodes_expanded > node_limit:
        print(">> Giving up (too many expansions)")
        return None

    max_fringe = max(max_fringe, len(frontier))
    f_curr, _, node = heapq.heappop(frontier)
    g_curr = node.cost
    state = node.state
    if g_curr > best_cost.get(state, float('inf')):
        continue
    pos, mask = state
    if mask == all_dots_mask:
        return {
            'path': reconstruct_multi_path(node),
            'cost': node.cost,
            'nodes_expanded': nodes_expanded,
            'max_depth': max_depth,
            'max_fringe': max_fringe
        }
    nodes_expanded += 1
    max_depth = max(max_depth, node.depth)
    for (nbr_pos, nbr_mask) in get_multi_neighbors(state, maze, goal_indices):
            new_state = (nbr_pos, nbr_mask)
            g2 = g_curr + 1
            if g2 < best_cost.get(new_state, float('inf')):
                best_cost[new_state] = g2
                h2 = heuristic_fn(new_state, goals, goal_indices)
                child = Node(state=new_state, parent=node, cost=g2, depth=node.depth+1, f=g2 + h2)
                counter += 1
                heapq.heappush(frontier, (child.f, counter, child))
    return None
```

End of Report