Shanat Barua - sbarua2
Umang Chavan - uchavan2
Varun Pitta - pitta2

MP1 Report

Part 1.1 - Basic Search

The aim of the first part of this report was to implement the following four search algorithms to solve given mazes:

1. Depth-First Search (DFS)
2. Breadth-First Search (BFS)
3. Greedy Best-First Search (GBFS)
4. A* Search (A*)

The tables below show relevant summary statistics for the implemented algorithms. Table 1 shows the solution path cost, that is the number of nodes it takes to reach from the defined start to final states based on the solution provided by the algorithm. It should be noted that in Table 1 the solution path costs include the final state of the algorithm. As expected BFS and A* deliver the optimal/shortest solutions to the maze for all mazes inputted. GBFS delivers an optimal solution only in the case of an open maze but still performs better than DFS which based on its algorithm will not deliver the optimal solution to the maze but rather just deliver any solution to the maze

Table 1: Solution Path Cost

	Medium Maze	**Big Maze**	**Open Maze**
Depth-First Search	117	257	192
Breadth-First Search	105	157	54
Greedy Best-First Search	121	163	54
A* Search	105	157	54

Table 2 shows the number of nodes that needed to be expanded to find a solution to the maze. Again, as expected BFS though it delivers the optimal solution it takes up the most memory space as it is not an informed search like GBFS or A* search, which use heuristics which help minimise the number of nodes expanded.

Table 2: Number of Nodes Expanded

	Medium Maze	**Big Maze**	**Open Maze**
Depth-First Search	51	84	228
Breadth-First Search	629	1256	557
Greedy Best-First Search	143	262	172

A* Search	457	1170	396

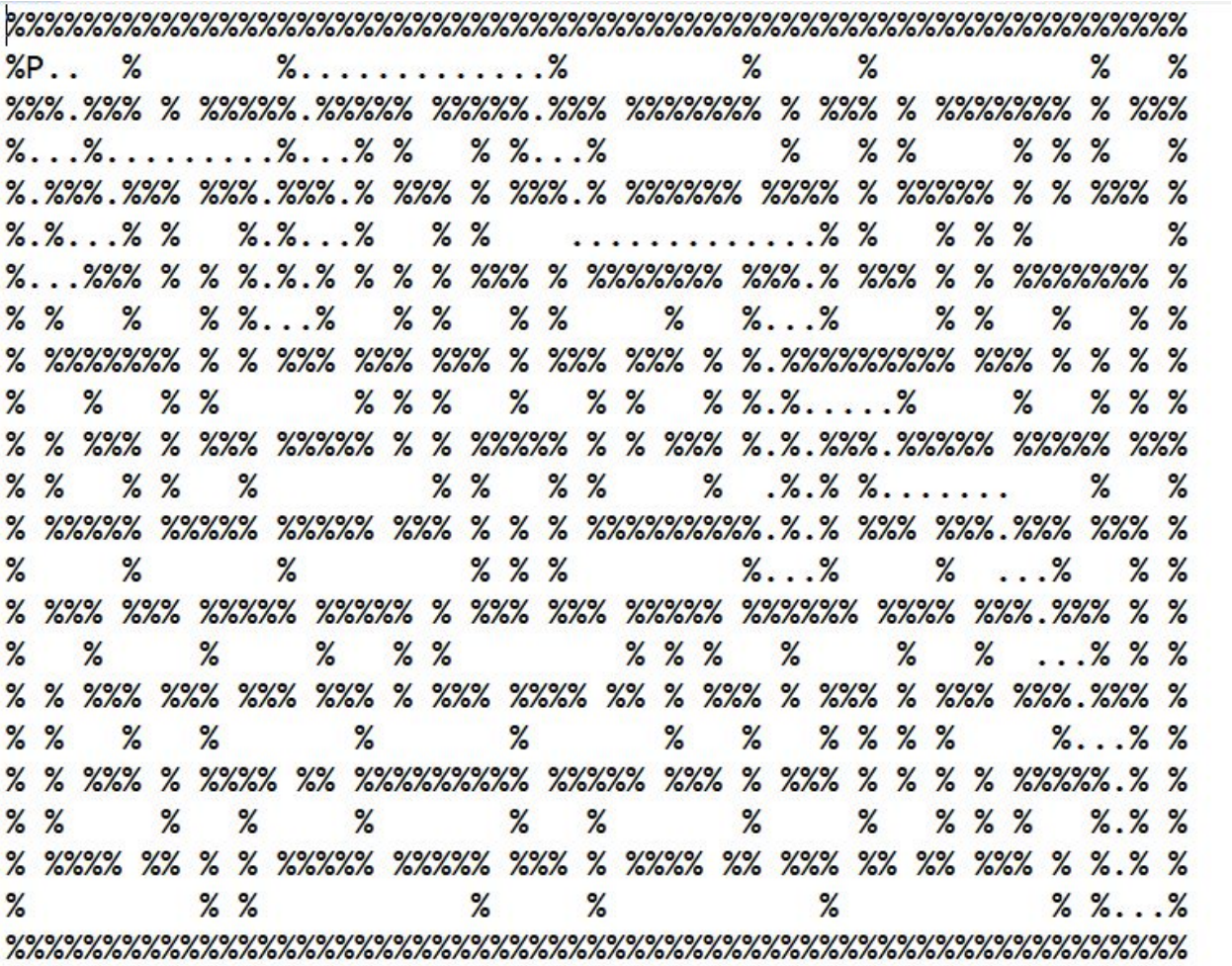
The algorithms implemented in Part 1.1 of this MP are described further in the sections below.

Depth-First Search (DFS)

DFS is a search algorithm which can be used to identify whether there is a path from the start state to the end state. It is primarily used in graph traversal, but in our case we have used it in maze traversal. Our maze could be represented as a graph as well, with a node being a space in the maze, and the edges being its immediate neighbors, but we did not choose to go that way. DFS, as the name suggests, goes as deep in one direction as possible before it reaches the solution or a wall. If it's a solution, we can return the contents of our stack, which will be holding the path from start to end. If it is a wall, we try a different direction and go as deep as possible in that direction to all the cells we have not yet visited. If we cannot proceed in any direction because we have been to the cell or the cell is a wall, we backtrack till the first cell in which we can go in another direction. This is the overall structure of DFS, which is also explained a bit more in depth in the comment section of our code. For our DFS implementation, we chose to make the sequence of directions right, left, up, down. Depending on how the sequence of directions was chosen, the solution cost and nodes expanded changes, but we decided to stick with right, left, up, down. In each of the mazes, we do not see an optimal solution, but we see a definite solution from the start state to the end state, which is what DFS guarantees if such a path exists. The output from the implemented DFS algorithm on the given mazes can be seen in the next three figures.

Shanat Barua - sbarua2
Umang Chavan - uchavan2
Varun Pitta - pitta2

Figure 1. DFS Medium Maze



Shanat Barua - sbarua2
Umang Chavan - uchavan2
Varun Pitta - pitta2

Figure 2. DFS Big Maze

```
Path cost: 257
Nodes expanded: 84
```

Shanat Barua - sbarua2
Umang Chavan - uchavan2
Varun Pitta - pitta2

Figure 3. DFS Open Maze

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                     %P............%
%                     %.............%
%                     %.............%
%                     %.............%
%                     %.............%
%                     %%%%%%%%......%
%                            %......%
%                            %......%
%                            %......%
%............................%......%
%......                 .....%......%
%......%%%%%%%%%%%%%%%%%............%
%......%                            %
%......%                            %
%......%                            %
%......%                            %
%......%%%%                         %
%     ....%                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Path cost: 192
Nodes expanded: 228
```

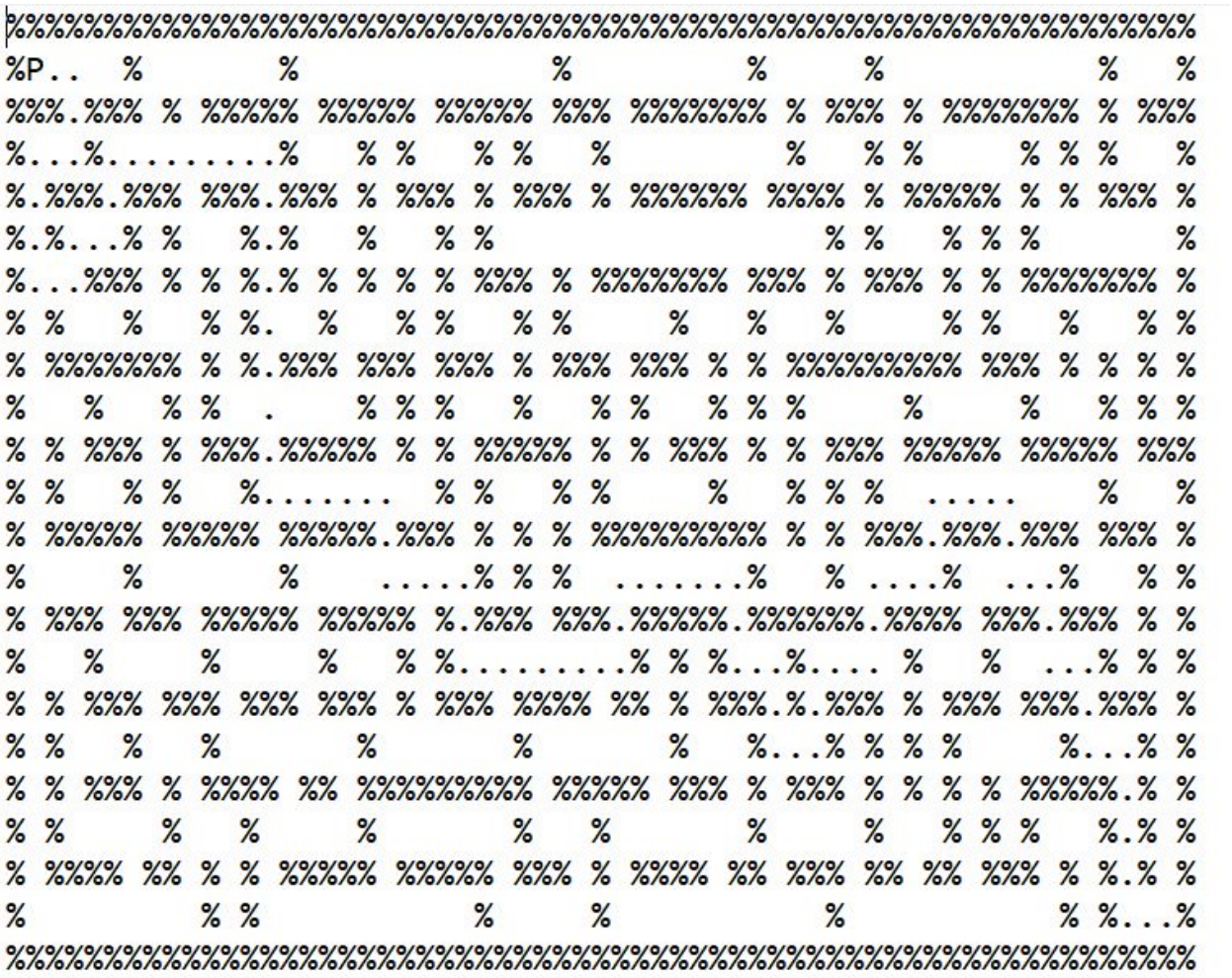
Shanat Barua - sbarua2
Umang Chavan - uchavan2
Varun Pitta - pitta2

Breadth-First Search (BFS)

BFS is an algorithm for traversing graph data structures. It begins with one node of the graph and then examines the children/neighbour nodes of the initial starting node before moving on to examine the neighbour nodes on the next level of the graph. The algorithm keeps doing this until it has examined all the nodes on the graph or reached a desired final state. The algorithm uses a queue to keep track of the nodes it has to examine. The algorithm also keeps track of the nodes it has examined to ensure that it doesn't visit the same node twice. If there is a start and end state that needs to be achieved this algorithm will always return the shortest/optimal path from the start to the end state for the the graph. The implementation of BFS in this MP is very similar to the traditional implementation of BFS. The start node in this implementation is the starting position of the Pac-man in the maze. The algorithm adds the starting node to the queue, dequeues this node and then examines and enqueues the neighbouring nodes (right node, left node, up node and down node) to the queue as long as they are not walls of the maze. The algorithm repeats this process of dequeuing the node it has examined and enqueueing its neighbors until it has found the end maze or examined all the nodes in the maze. Unlike DFS which picks a direction of nodes and follows it until it has hit a dead end or found the solution, BFS examines all the nodes at one level of graph before moving onto the next level. This ensures the shortest solution path but it also means that this algorithm will take up the most memory/expand the most nodes to find this path as seen in Table 2. The commented code can be referred to get a better understanding of the implemented algorithm. The output from the implemented BFS algorithm on the given mazes can be seen in the next set of figures.

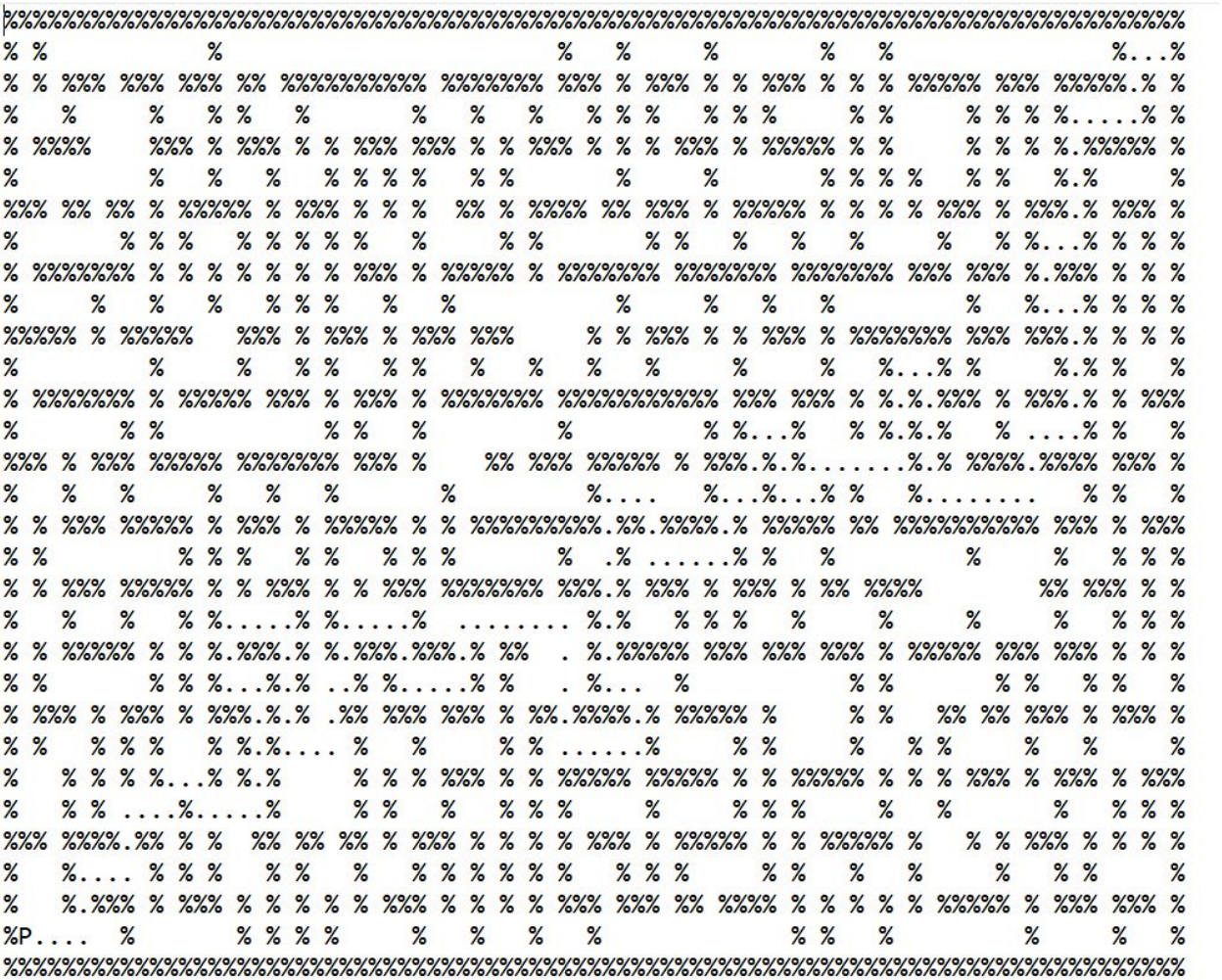
Shanat Barua - sbarua2
Umang Chavan - uchavan2
Varun Pitta - pitta2

Figure 4. BFS Medium Maze



Shanat Barua - sbarua2
Umang Chavan - uchavan2
Varun Pitta - pitta2

Figure 5. BFS Big Maze



Shanat Barua - sbarua2
Umang Chavan - uchavan2
Varun Pitta - pitta2

Figure 6. BFS Open Maze

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                     %P.......     %
%                     %       .     %
%                     %       .     %
%                     %       .     %
%                     %       .     %
%                     %%%%%%%%.     %
%                            %.     %
%                            %.     %
%                            %.     %
%                            %.     %
%     ...................    %.     %
%     .%%%%%%%%%%%%%%%%%.......     %
%     .%                            %
%     .%                            %
%     .%                            %
%     .%                            %
%     .%%%%                         %
%     ....%                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Path cost: 54
Nodes expanded: 557
```

Shanat Barua - sbarua2
Umang Chavan - uchavan2
Varun Pitta - pitta2

Greedy Best-First Search (GBFS)

Greedy Best-First Search is an algorithm similar to BFS, but it uses a Priority Queue structure with the specific intent of picking the “smallest” element. In our case, the heuristic we used was euclidean distance from the current cell to the final cell. The algorithm starts by calculating the distance to goal for every neighbor of the current cell. We then create a list to hold tuples of these distances and the corresponding cell. For example, a tuple for the “right” neighbor of the current cell would be (distance_to_goal_right, curr_cell_right). We then sort this list by distance in ascending order, and push the elements in the same order into our priority queue. With each new iteration, the element with the smallest distance to the goal is popped and the same process continues. Greedy BFS is similar to BFS in the sense that it pushes on all of the corresponding neighbors, but it expands far less nodes since it is seeking the neighbors closest to the goal first, instead of choosing a neighbor at random. Greedy BFS does not guarantee the shortest path, however. Since its heuristic is distances, it does not look ahead to see if there is a wall blocking the path. Therefore once it hits the wall, it would have to make its way around it to the solution, instead of potentially choosing a path that avoids the wall altogether. The algorithm is explained in more detail in the comments section of the code and this same explanation can be found below as well. It is encouraged to view the following paragraph in context to the accompanying code to gain the best understanding of the implemented algorithm.

Greedy Best First Search has been implemented for this MP using a euclidean distance heuristic. Euclidean distance is implemented in the Distance() function using the following formula

$$Distance = \sqrt{(X2 - X1)^2 + (Y2 - Y1)^2}$$

A priority queue is used to maintain organize the queue in ascending order by distance to the goal. Starting from the start index, the algorithm will push on indices of the current cell's neighbors in order of distance to the goal state (smallest distance to goal pushed on first). The algorithm will then pop off the neighbor with the smallest distance to the goal first, and then push on its neighbors in a similar fashion. The path is kept track with the parent dictionary, which is useful in the backtrace() function. Once the algorithm recognizes the goal state cell, it breaks from the loop and calls on backtrace to highlight the path. The output for the algorithm as applied to the maze can be seen in the figures below.

Figure 7. Greedy Medium Maze

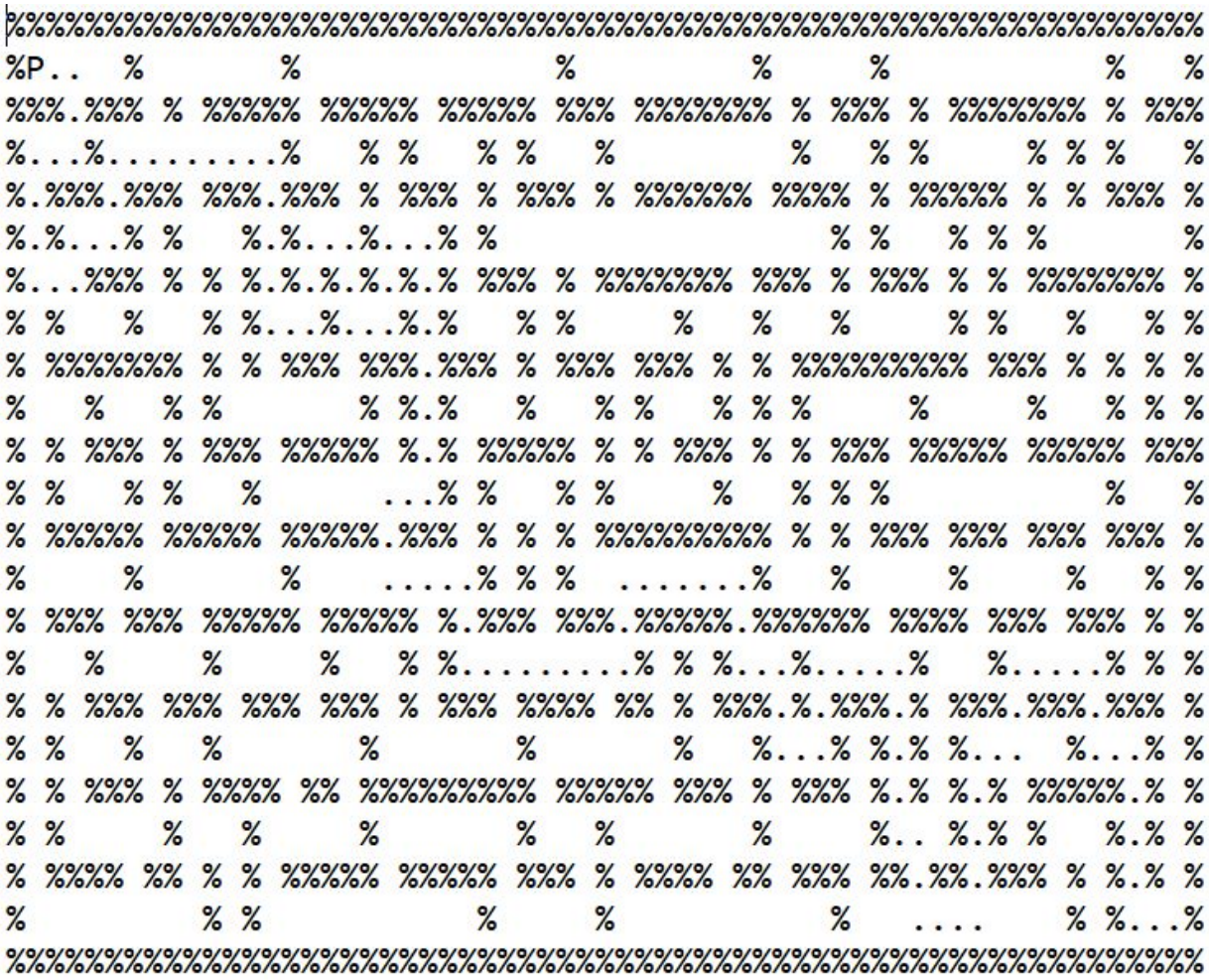
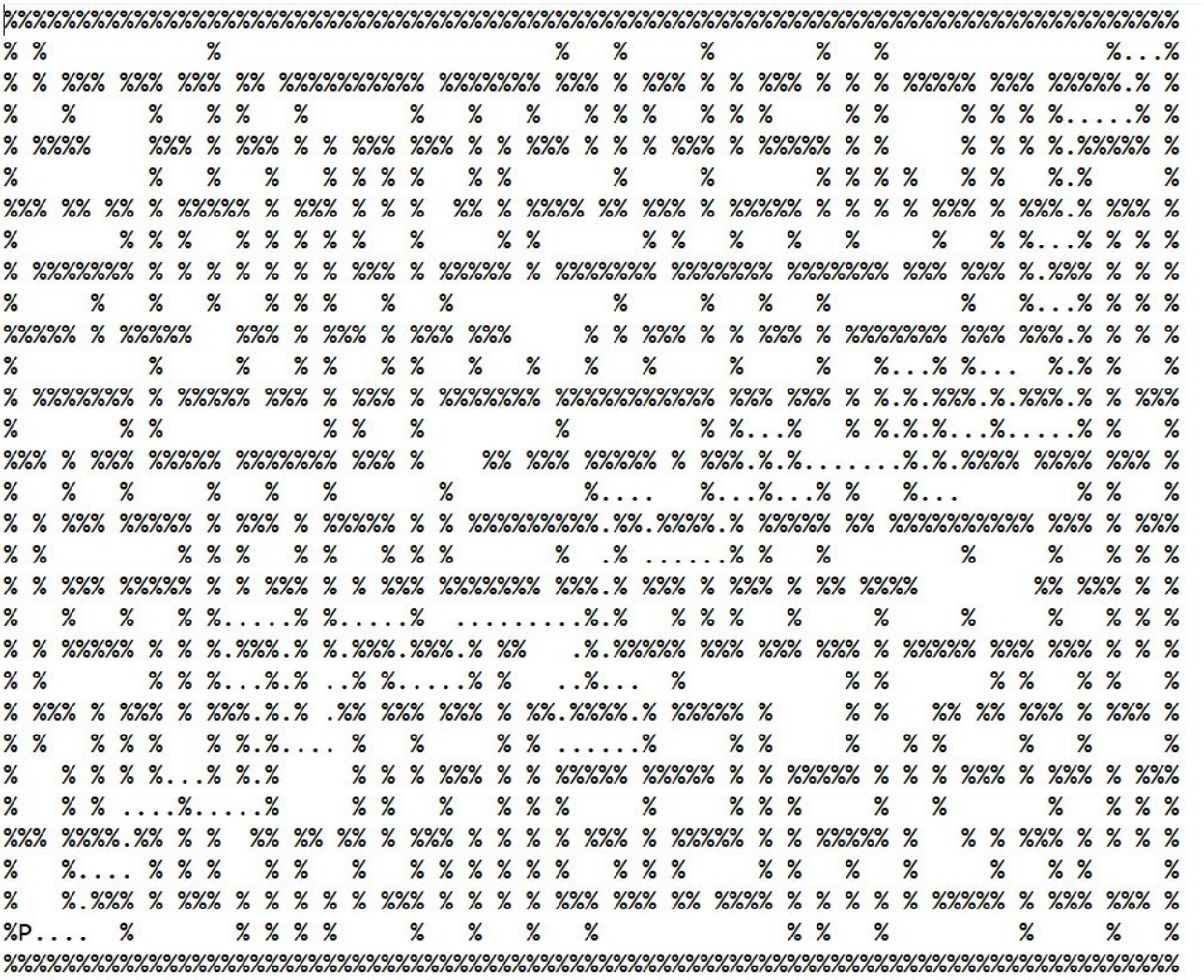


Figure 8. Greedy Big Maze



```
Path cost: 163
Nodes expanded: 262
```

Figure 9. Greedy Open Maze

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                     %P            %
%                     %.            %
%                     %.            %
%                     %.            %
%                     %........     %
%                     %%%%%%%%.     %
%                            %.     %
%                            %.     %
%                            %.     %
%                            %.     %
%     ...................    %.     %
%     .%%%%%%%%%%%%%%%%%.......     %
%     .%                            %
%     .%                            %
%     .%                            %
%     .%                            %
%     .%%%%                         %
%     ....%                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Path cost: 54
Nodes expanded: 172
```

Shanat Barua - sbarua2
Umang Chavan - uchavan2
Varun Pitta - pitta2

A* Search

The A* Search algorithm is another example of a prioritized search. The difference about this algorithm compared to DFS/BFS is that it employs informed search strategies. The main idea is that as the algorithm runs, each step provides information about previous steps so that the desirable state may be entered through a different transition. We accomplish this behavior by using an evaluation function to rank nodes and selecting the most desirable state to expand and move on with. The evaluation function is the estimated total cost of the path through a given node **n** to the goal state. It is defined as:

$$f(n) = g(n) + h(n)$$

This evaluation function is calculated at each node along the way to the goal to determine what the optimal solution is. The g(n) term, also known as the path cost, is the cost so far to reach a given node **n**. The h(n) term, also known as the heuristic, is the estimated cost from the current node **n** to the goal state.

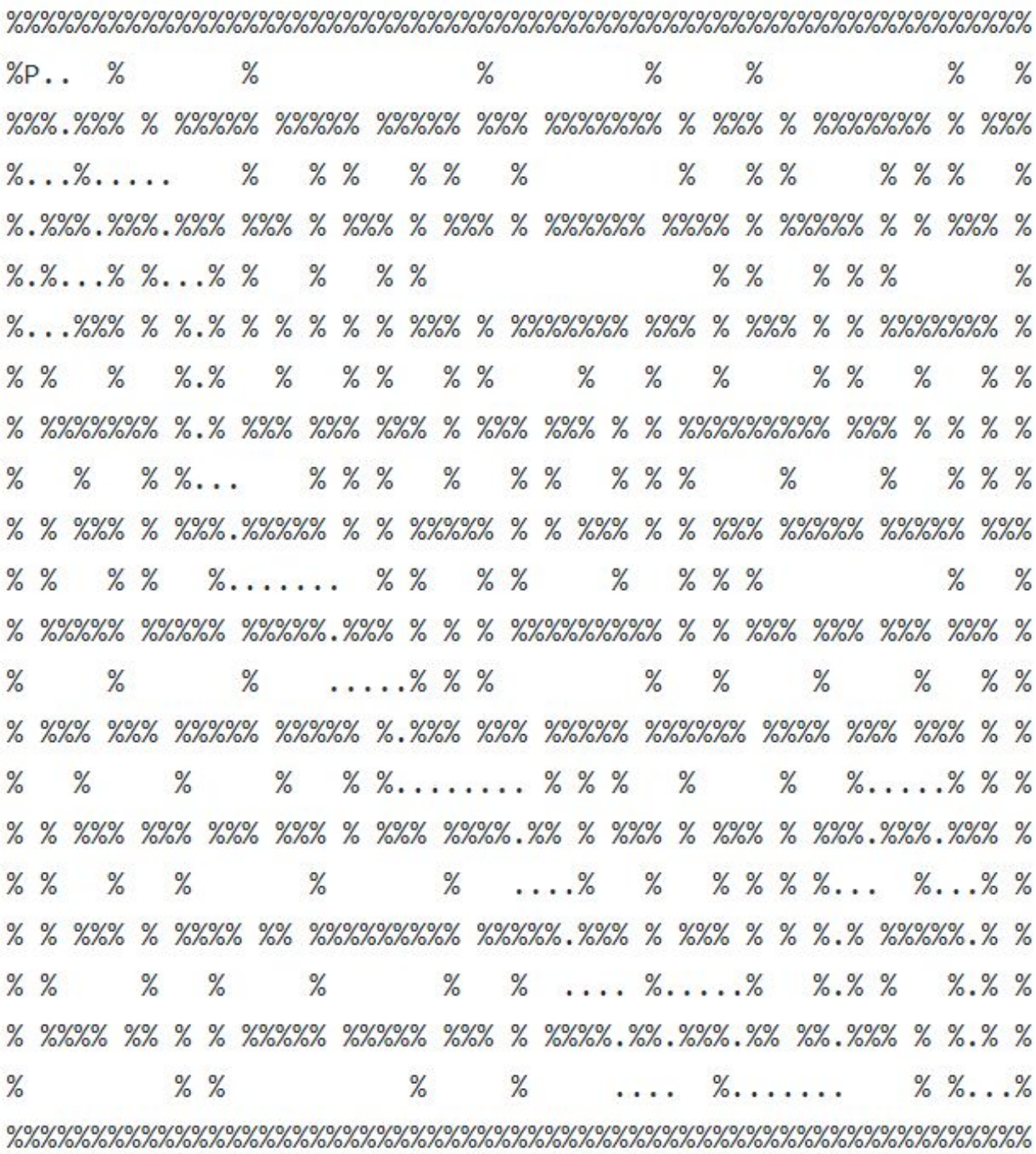
Keeping track of g and h at each step of the process is very important because this is the way that the A* algorithm decides which is the most optimal path to take. In order for the algorithm to be optimal, the heuristic that we decide to use has to be admissible which means that the measurement never overestimates the cost to reach the goal. For 1.1, we used the Manhattan distance as the heuristic because it is admissible and easier to use when dealing with walls in mazes. The Manhattan Distance between two points p1(x1,y1) and p2(x2, y2) is found using the following equation:

$$\textit{Manhattan Distance (p1, p2)} = \text{abs}(x1-x2) + \text{abs}(y1-y2)$$

The algorithm works by pushing the starting node onto a MinHeap and then running a loop until the goal state is found. The loop pops the smallest cost term off the MinHeap, gets its neighbors and then updates their costs. If the child has already been added to the Heap, then we ignore it unless the new cost is less than the one currently on the heap. We update the node and place it back on the heap. This process repeats until the goal state is achieved. Shown below are the diagrams associated with the solved A* algorithm.

The following cost of paths do not include the end node and are hence one less than those in Table 1.

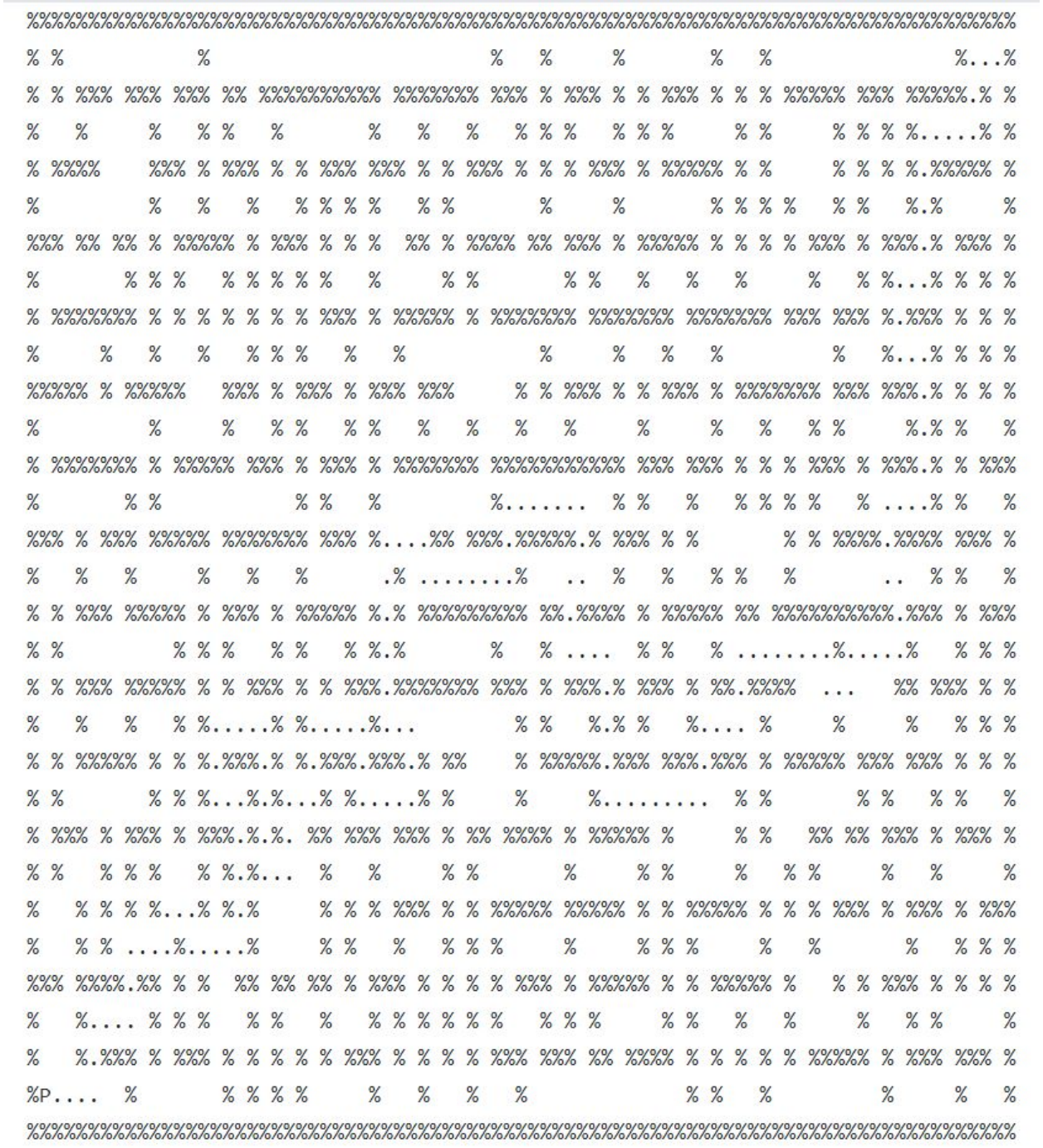
Figure 10. A* Medium Maze



Cost of Path: 104

Nodes Expanded: 457

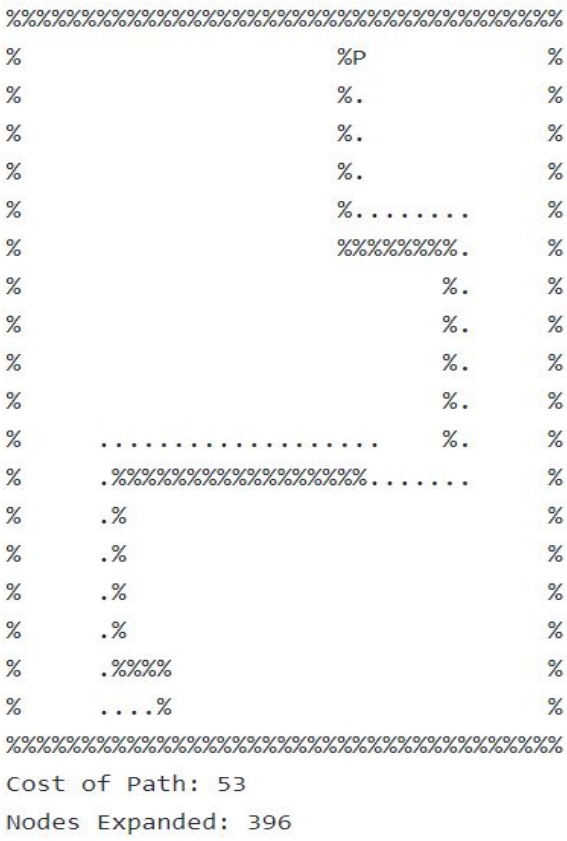
Figure 11. A* Big Maze



Cost of Path: 156

Nodes Expanded: 1170

Figure 12. A* Open Maze



Part 1.2 - Search with multiple dots

Part 1.2 was very similar to 1.1 in that we are still using the A* algorithm to achieve the solution. The difference here, however, is that we were trying to run the algorithm with multiple goal states. Because of this, the previous heuristic we had would not be strong enough to calculate the optimal path for capturing all pellets. The heuristic we used to achieve this was the length of the Minimum Spanning Tree rooted at a given node. This gives a better admissible heuristic than the usual Manhattan distance. The same overall algorithm is the same from 1.1, except this time, we generate an MST as the heuristic. We create a lookup table that acts as the weighted graph between each possible node in this maze. Instead of expanding each individual node in the graph, we expand on just the list of goal states. This allows to cut down on a lot of the unnecessary traversals. The table below shows the path costs and the number of expanded nodes on the solution path.

Table 3. Solution Path Cost & Nodes Expanded for Search with Multiple dots

	Tiny Search	**Small Search**	**Medium Search**
Solution Path Cost	75	197	452
Number of Nodes Expanded	118	429	1554

The following images show the solutions of the path that is taken when trying to collect all the pellets.

Figure 13. Search with multiple dots - Tiny search

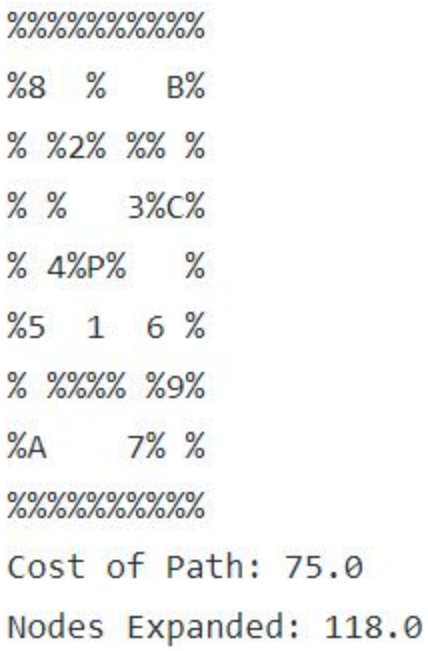


Figure 14. Search with multiple dots - Small search

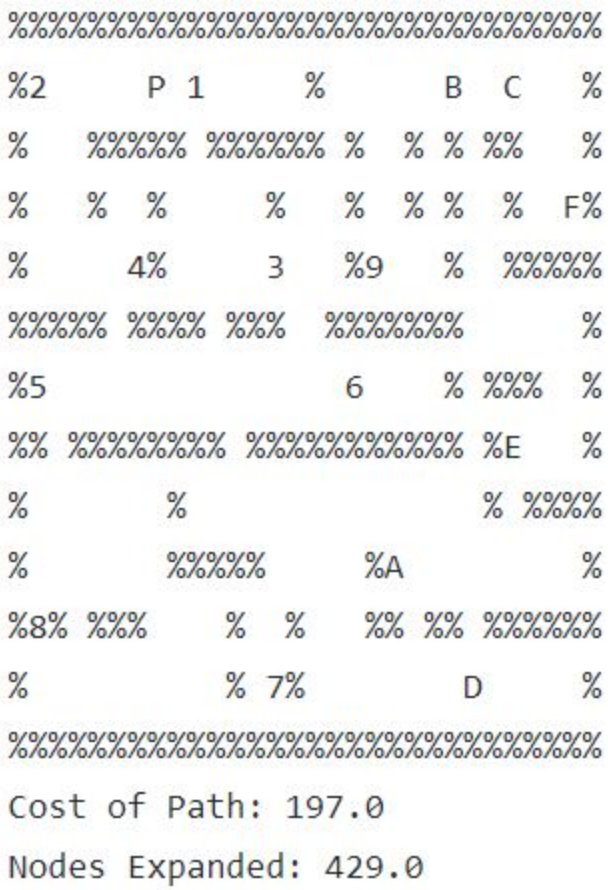
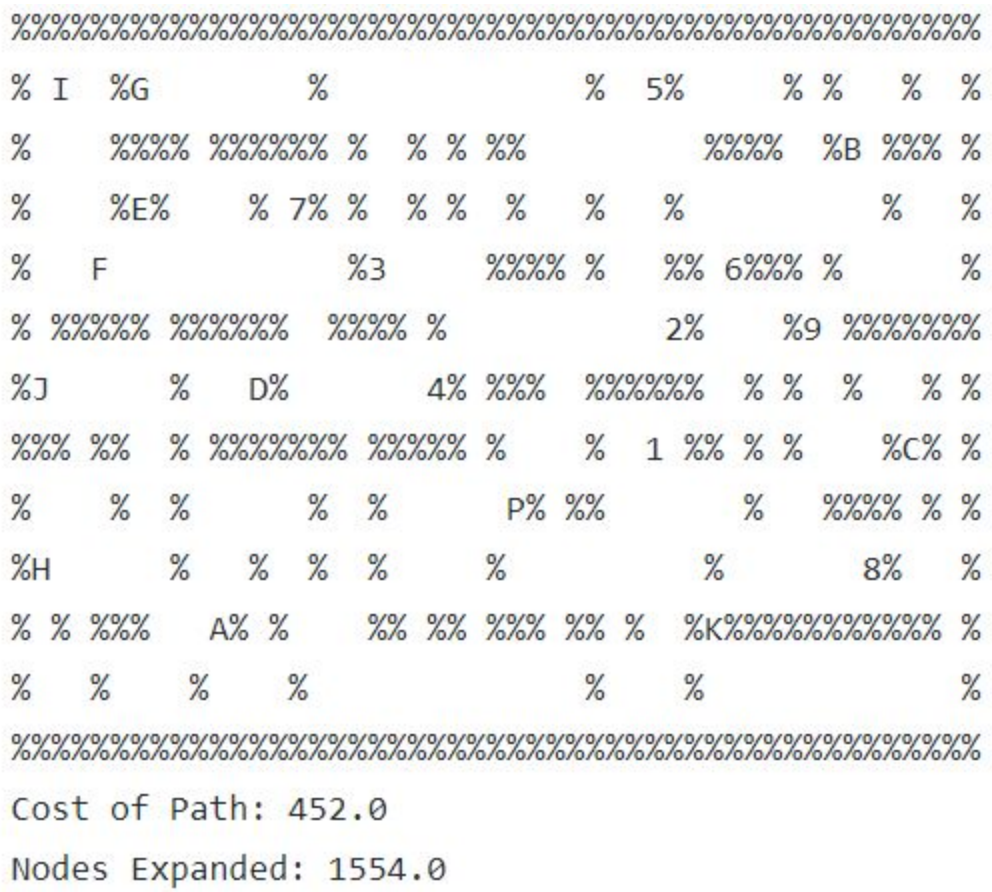


Figure 15. Search with multiple dots - Medium search



Contribution of Work

1. Shanat Barua - BFS, Report, Reviewed Report
2. Umang Chavan - Maze, DFS, GBFS, Report
3. Varun Pitta - A*, Search with multiple dots, Report