Assignment Report

Course name: Applied AI - DV2659
Teacher: Dr. Huseyin Kusetogullari
Institution: Blekinge Institute of Technology (BTH)
Degree Programme: Master's Programme in AI and Machine Learning
Submitted By: Muhammad Umair Akram Butt

Date: September 14, 2025

Table of Contents

List of Figures

List of Tables

Assignment 1 – Agents, Graph, Graph Searches, and Implementation

1. Problem 1: Shopping Assistant Agent

The following is the implementation of a shopping assistant using a simple reflex agent and an advanced utility-based agent.

1.1. Part A: Reflex Agent

A reflex agent is a simple agent and does not do any planning. It just makes a purchase decision (buy or skip) by analyzing a single item's properties from the grocery list by doing a budget vs. price comparison. It does not consider past purchases or goal attainment.

Implementation details: The starting state is quantitatively described in section 1.1.1; the end state is qualitatively described in section 1.1.2. We will reach the end state by applying a set of rules described in section 1.1.3. While section 1.1.4 shows the implementation of the reflex agent, and lastly, in section 1.1.5, the final state is shown.

1.1.1. Starting State

budget = 20
basket = [] (empty)
overall_satisfaction = 0

1.1.2. End State:

Exhaust the budget or evaluate all the items in the grocery list.

1.1.3. Condition-Action Rules

The independent variable or the deciding factor is the cost of the reflex agent.

```
# Until we exhaust the budget or evaluate all the items in the
# grocery list, keep on applying the condition-action rules

while (budget != 0) or (len(grocery_list) != 0):
   if price < budget:
         item →buy
         budget → budget - price
         overall_satisfaction += satisfaction
   if price > budget:
         skip item
```

1.1.4. Implementation:

The implementation detail is shown in Table 1.

Table 1: Implementation details of Reflex Shopping Agent

Sr. #	Percept Sequence	Action	Current State

1.	{
	'item': 'bread',
	'price':5,
	'satisfaction': 6
	}
		Buy	budget = 15
			basket = ['bread']
			overall_satisfaction = 6

2.	{
	'item': 'milk',
	'price':4,
	'satisfaction': 7
	}
		Buy	budget = 11
			basket = ['bread', 'milk']
			overall_satisfaction = 13

3.	{
	'item': 'eggs',
	'price':6,
	'satisfaction': 8
	}
		Buy	budget = 5
			basket = ['bread', 'milk', 'eggs']
			overall_satisfaction = 21

4.	{
	'item': 'chocolate',
	'price':8,
	'satisfaction': 9
	}
		Skip	unchanged

1.1.5. End State:

After applying the defined rules, here is the final end state.

budget = 5
basket = ['bread', 'milk', 'eggs']
overall_satisfaction = 21

1.2. Part B: Utility-Based Agent

In contrast to a reflex agent that acts greedily, a utility-based agent acts more strategically. Its aim is to achieve a well-defined goal while maximizing utility or satisfaction.

1.2.1. Utility Function

Maximize the satisfaction without exhausting the $20 budget. Mathematically, we can represent it like this:

$$Maximize\ U(basket) = \sum_{items\ in\ bucket} satisfaction$$

Constraints:

$$\sum_{items\ in\ bucket} price \leq 20$$

Section 1.2.1 defines the utility function that is the end state we want to achieve using our utility-based agent. We are going to create different combinations of the items and select the best option based on the utility function. For creating these combinations, in addition to the already available data points, we are going to create a new data point (heuristic), the satisfaction-price ratio. After creating these combinations, we will select one (or more, if possible) basket that maximizes the satisfaction.

1.2.2. Satisfaction-Price Ratio

We can calculate this ratio using this formula:

$$satisfaction - price\ ratio = \frac{satisfaction}{price}$$

Table 2 shows the value of this ratio for each item. It is important to highlight that this ratio is not the sole deciding factor in basket formation, and it will only be used for creating different baskets, along with other criteria.

Table 2: Satisfaction-Price Ratio of all the items in the grocery list

Sr. #	Item	Price	Satisfaction	Satisfaction-Price Ratio
1.	Bread	5	6	1.2
2.	Milk	4	7	1.75
3.	Eggs	6	8	1.333
4.	Chocolate	8	9	1.125

1.2.3. Basket Combinations

This section chalks out all the various basket possibilities by combining different items from the grocery list. Table 3 shows various basket options (not an exhaustive list) with their price and satisfaction scores.

Table 3: Various basket options with their price and satisfaction values

Sr. #	Items in the Basket	Price Tag of the Basket	Satisfaction of the Basket
1.	Milk-Eggs-Bread	15	21
2.	Chocolate-Eggs-Milk	18	24
3.	Bread-Eggs-Chocolate	19	23
4.	Bread-Milk-Eggs-Chocolate	~~23 (out of budget)~~	~~30~~
5.	Eggs-Chocolate	14	17

1.2.4. Optimal Basket Selection

From Table 3, it is clearly evident that the second bucket provides the highest satisfaction. So, our utility-based agent will buy Milk, Eggs, and Butter, as it provides:

- Total Satisfaction: 24 (highest possible)
- Total Price: $18 (under budget)

1.2.5. Comparison With Reflex Agent

A utility-based shopping agent, in contrast with a reflex shopping agent, acts strategically. The reflex shopping agent was a greedy agent and focused narrowly on a predefined set of rules to aid the shopper. It did not evaluate past actions or future states. Moreover, it had no final goal to achieve.

Our utility-based agent, on the other hand, strived to maximize the value addition for the shopper. It chalked out all the various scenarios and selected the optimal shopping list that maximizes the utility for the shopper.

2. Problem 2: Graph Path Finding Using BFS and DFS Algorithms

2.1. Implementation Details:

The first step was to translate the given maze into a graph. For that, finding the edge connections was a bit challenging. So, I drew the maze and manually documented the edges. Figure 1 shows the screenshot of the manual calculations.

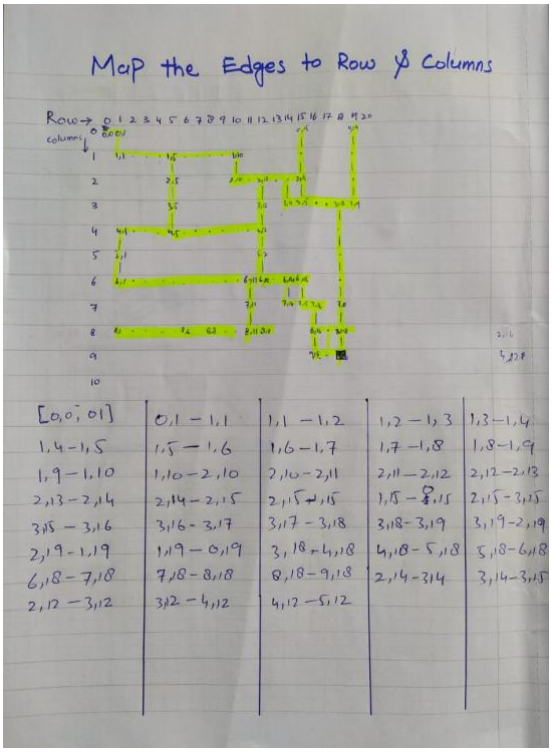

Figure 1: Manual calculations for translating the maze into edges

The resultant maze is stored in the maze.json file. This maze file was loaded into the main.py file to apply the BFS and DFS algorithms. There are three main functions:

- find_path_with_dfs → Applies the DFS algorithm to the input maze.
- find_path_with_bfs → Applies the BFS algorithm to the input maze.
- plot_maze_and_path → Can plot the original maze as well as the path solution created from either of the above functions, if provided.

Then there is the *main* function that loads the maze.json file and calls the aforementioned three functions to find the solution and plot it. The whole flow of the main function is shown in the following flowchart (Figure 2).

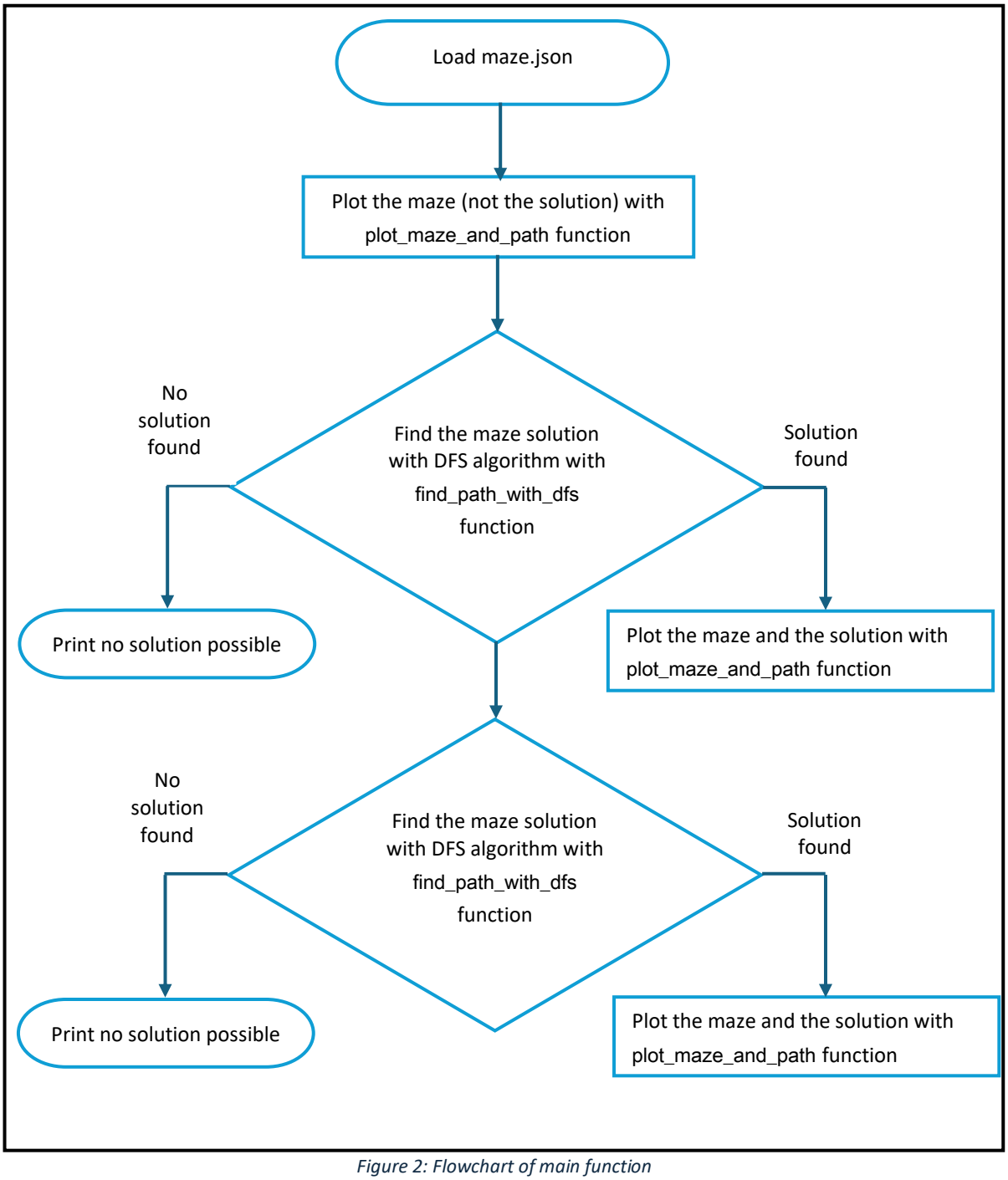

Figure 2: Flowchart of main function

2.2. Algorithm Implementation

The original maze that we need to solve is shown in Figure 3. The starting point is shown in red, while the goal is shown in blue.

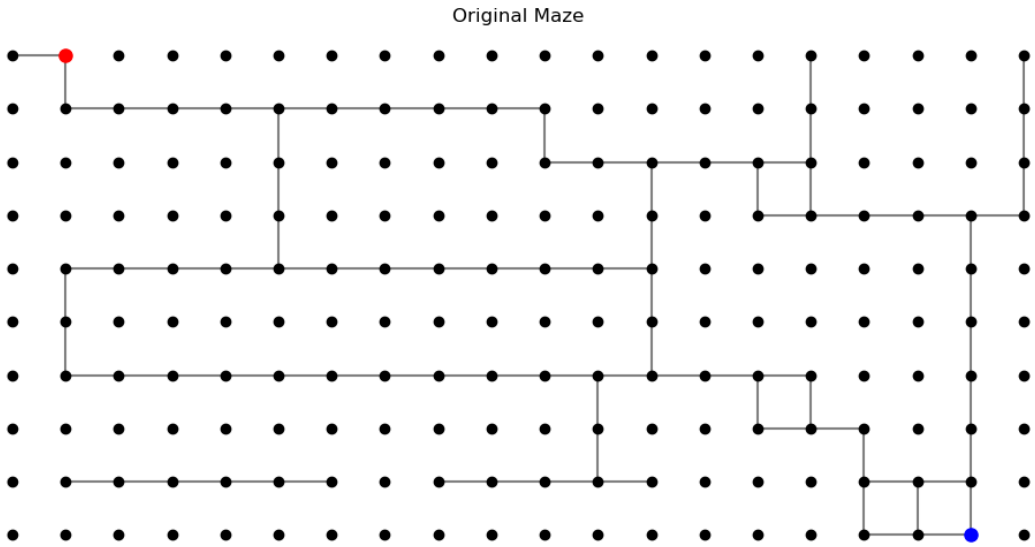

Figure 3: Original maze

2.2.1. Depth-First Search (DFS)

The DFS algorithm is implemented iteratively using a stack (LIFO - Last-In, First-Out). In DFS, for any node, we need to explore as deeply as possible along each branch before backtracking.

2.2.2. Breadth-First Search (BFS)

The BFS algorithm is implemented iteratively using a queue (FIFO - First-In, First-Out). As opposed to DFS, we need to explore each node and its branches level by level before moving to the next node on the next level.

2.3. Performance comparison of DFS and BFS

Figure 4 shows the path generated by DFS, and Figure 5 shows the path generated by BFS.

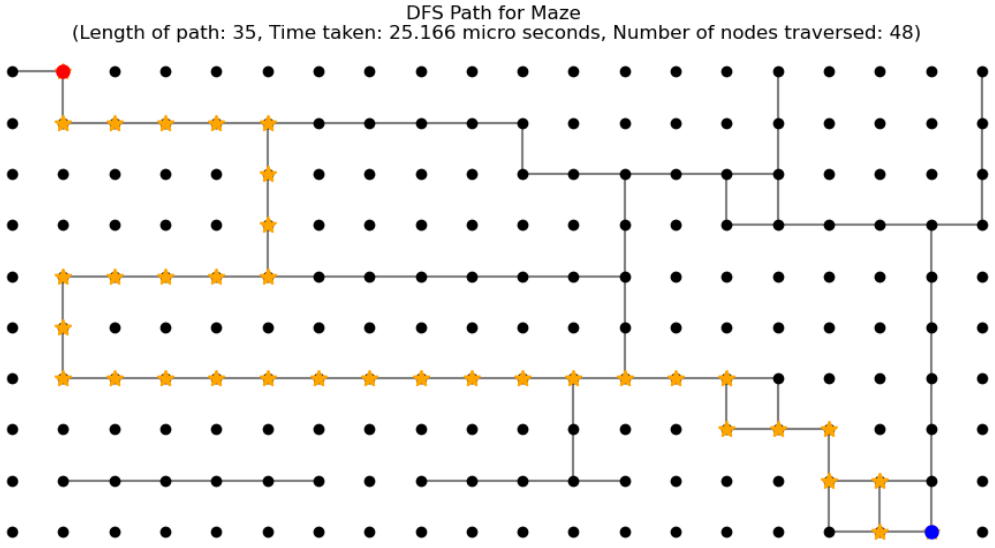

Figure 4: Path generated by DFS

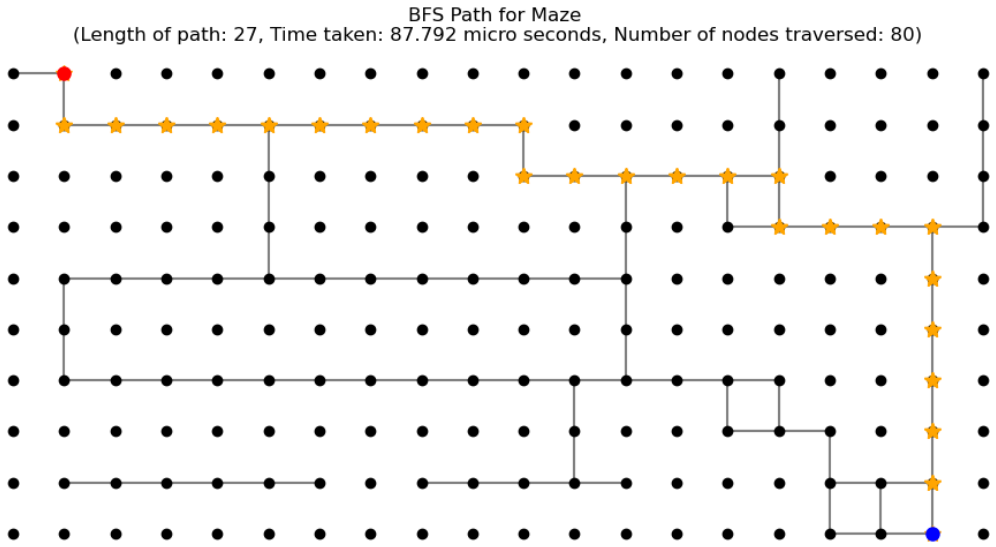

Figure 5: Path generated by BFS

From the graph, it is clearly evident that BFS generated the shortest path. Anyhow, the path is not the only evaluation metric. We need to compare the results based on:

- **Path length (optimality)**: The length of the path.
- **Number of nodes traversed to find the solution (performance)**: The number of nodes the algorithm analyzed to generate the resultant path.
- **Execution time (time efficiency)**: Total time the algorithm took to find the solution.

Table 4 summarizes the value of these metrics for each algorithm.

Table 4: Evaluation metric values for DFS and BFS algorithms

Sr. #	Evaluation Metric	DFS	BFS
1.	Path length (optimality)	35	27
2.	Number of nodes traversed to find the solution (performance)	48	80
3.	Execution time (time efficiency)	25.16 μ seconds	87.79 μ seconds

From the above stats, it is clear that:

- BFS is more optimal as it always gives the shortest path, but it is not time-efficient in our case. Because it took more time to find the solution. As far as performance is concerned, it also traverses through more nodes compared to DFS.

  Because our graph is unweighted, BFS is guaranteed to find the shortest possible path by exploring level by level. However, this comes at a higher computational cost compared to DFS. It is important to note that these observations about time efficiency and performance are not generally true; they apply specifically to the maze problem in our

case[1]*.*

- DFS does not guarantee the shortest path, but in our case, it found a solution more quickly. It is also noteworthy that it explores fewer nodes compared to BFS. Although it is not optimal, it is computationally less expensive because it does not need to explore the graph level by level. Instead, it goes deeper along one path before backtracking, which can lead to faster results in practice.

  These observations about speed and node exploration are specific to our maze and may not hold in all cases[2]*.*

[1] This observation is from AI. When I checked the accuracy of my observations, it stated that my observations related to time and number of nodes traversed are not generally true in all scenarios. BFS checks all nodes at depth ≤ d.

[2] This observation is also from AI when I check the validity of my observations.