**Level 1 – Basic Understanding**

**🔹 Task 1: Define a Search Problem**

define:

* Initial state
* Possible actions
* Goal test
* Path cost

for

* Maze
* City map (from A to B)
* Robot vacuum cleaner

➡️ **Classify** if it is:

* Goal-based?
* Deterministic?
* Observable?

1. Maze

* • Initial State:  
  The starting position of the agent (e.g., cell (1,1) in the maze grid).
* • Possible Actions:  
  Move up, down, left, or right (assuming no walls block the path).
* • Goal Test:  
  Check if the agent has reached the goal cell (e.g., cell (5,5)).
* • Path Cost:  
  Usually each move has a cost of 1. Total cost = number of steps taken.
* 🔹 2. City Map (From A to B)
* • Initial State:  
  Starting city or location (e.g., city A).
* • Possible Actions:  
  Travel to directly connected cities via available roads.
* • Goal Test:  
  Check if the current city is the destination city (e.g., city B).
* • Path Cost:  
  Cost could be distance in kilometers, travel time, or fuel used between cities.
* 🔹 3. Robot Vacuum Cleaner
* • Initial State:  
  The robot’s starting position and current map of clean/dirty tiles.
* • Possible Actions:  
  Move in one of four directions (up, down, left, right), and clean the current tile.
* • Goal Test:  
  All dirty tiles have been cleaned.
* • Path Cost:  
  Could be the number of actions (moves + cleans) or energy consumed.

🔹 1. Maze

| Property | Classification |
| --- | --- |
| Goal-based | ✅ Yes |
| Deterministic | ✅ Yes |
| Observable | ✅ Yes (assuming full map is known) |

* 🔹 2. City Map (from A to B)

| Property | Classification |
| --- | --- |
| Goal-based | ✅ Yes |
| Deterministic | ✅ Yes |
| Observable | ✅ Yes (if the map and traffic are known and fixed) |

* 🔹 3. Robot Vacuum Cleaner

**Task 2: BFS vs DFS Comparison Table**

**Fill in a table comparing:**

* **Data structure used**
* **Time and space complexity**
* **Completeness**
* **Optimality**
* **When to use**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | Feature | BFS (Breadth-First Search) | DFS (Depth-First Search) | | --- | --- | --- | | Data Structure Used | Queue | Stack (or recursion) | | Time Complexity | O(b<sup>d</sup>) | O(b<sup>m</sup>) | | Space Complexity | O(b<sup>d</sup>) | O(b·m) (linear with depth) | | Completeness | ✅ Yes (if branching factor *b* is finite) | ❌ No (may go down infinite paths) | | Optimality | ✅ Yes (if all step costs are equal) | ❌ No (returns first found path) | |  |  |  | |

**Task 3: Manual BFS and DFS Tracing Draw a binary tree (depth = 3).**

**For BFS:**

* **List visited nodes at each level**
* **Track queue at each step**

**For DFS:**

* **List order of node visits**
* **Show stack at each step**

🔷 Binary Tree (Depth = 3)

A

/ \

B C

/ \ / \

D E F G

🌳 Breadth-First Search (BFS)

BFS uses a queue (FIFO) and visits nodes level by level.

✅ Visited Nodes at Each Level:

* Level 0: A
* Level 1: B, C
* Level 2: D, E, F, G

🧾 BFS Queue at Each Step:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Step** | **Visited Node** | **Queue** | | --- | --- | --- | | 1 | A | [B, C] | | 2 | B | [C, D, E] | | 3 | C | [D, E, F, G] | | 4 | D | [E, F, G] | | 5 | E | [F, G] | | 6 | F | [G] | | 7 | G | [] | |

DFS Stack at Each Step:

| **Step** | **Visited Node** | **Stack** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| 1 | A | [C, B] |

|  |  |  |
| --- | --- | --- |
| 2 | B | [C, E, D] |

|  |  |
| --- | --- |
| 3 | D [C, E] |

|  |  |  |
| --- | --- | --- |
| 4 | E | [C] |

|  |  |  |
| --- | --- | --- |
| 5 | C | [G, F] |

|  |  |  |
| --- | --- | --- |
| 6 | F | [G] |

|  |  |
| --- | --- |
| 7 | G |

[]

**🔁 Level 2 – Code Implementation**

**🔹 Task 4: Write BFS and DFS in Python**

Use an **adjacency list** representation.

* Find a goal node in a graph
* Trace visited nodes

Add:

* Print statement to show queue/stack
* Print visited nodes in order

|  |  |  |
| --- | --- | --- |
|  |  |  |