Assignment for Search Strategies

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?

🔹 Task 2: BFS vs DFS Comparison Table

Fill in a table comparing:

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

🔹 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

🔁 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

🔹 Task 5: Add Depth-Limited DFS (DLS)

Modify your DFS code to support depth-limiting.

* Try depth = 2, 3
* Observe how it avoids infinite loops

🔹 Task 6: Compare Time and Space Complexity

For a binary tree of depth 5:

* Count nodes visited in BFS and DFS
* Estimate time complexity O(b^d)
* Estimate space complexity

🧠 Level 3 – Problem Solving & Challenges

🔹 Task 7: Maze Solver with BFS and DFS

Design a 2D grid maze (start to goal). Use:

* 0 for free cell, 1 for wall
* Start = (0,0), Goal = (n-1,n-1)

Implement:

* BFS to find shortest path
* DFS to find any path

Add visualization:

* Print grid with path traced

🔹 Task 10: Design Your Own Search Problem

Come up with a real-world inspired search problem. Examples:

* AI agent in a warehouse
* Navigation on a game map
* Package delivery route optimization

Define:

* State representation
* Action model
* Goal test
* Search strategy you'd use and why

Let's break down the definitions for each component and classify the environments.

1. Maze

- Initial state: The starting position of the agent in the maze.

- Possible actions: Moving up, down, left, or right (assuming a grid-based maze) to adjacent cells.

- Goal test: Reaching the exit or a specific target position in the maze.

- Path cost: The number of steps taken or the distance travelled to reach the goal.

Classification:

- Goal-based: Yes, the agent has a specific goal to reach the exit.

- Deterministic: Yes, the outcome of actions is certain (assuming no random elements like moving obstacles).

- Observable: Yes, the agent can observe its current position and the maze layout.

2. City Map (from A to B)

- Initial state: The starting location A.

- Possible actions: Moving along roads to adjacent intersections or locations.

- Goal test: Reaching the destination B.

- Path cost: Distance, time, or another metric like fuel consumption.

Classification:

- Goal-based: Yes, the goal is to reach location B from A.

- Deterministic: Generally yes, assuming predictable traffic patterns and no random events.

- Observable: Yes, the agent can observe its current location and map.

3. Robot Vacuum Cleaner

- Initial state: The starting position and state (clean/dirty) of each room.

- Possible actions: Moving to an adjacent room and cleaning a room if it's dirty.

- Goal test: All rooms are clean.

- Path cost: The number of movements and cleaning actions.

Classification:

- Goal-based: Yes, the goal is to clean all rooms.

- Deterministic: Yes, the outcome of actions (moving and cleaning) is predictable.

- Observable: Yes, the robot can observe the state (clean/dirty) of the rooms it's in or has access to the state of all rooms.

These classifications assume ideal conditions. Real-world scenarios might introduce uncertainties, making the environment partially observable or stochastic. Let me know if you'd like further details!

Here's a comparison table for Breadth-First Search (BFS) and Depth-First Search (DFS):

|  |  |  |
| --- | --- | --- |
| Criteria | Breadth-First Search (BFS) | Depth-First Search (DFS) |
| Data Structure | Queue | Stack (or recursion) |
| Time Complexity | | O(|V| + |E|) (vertices + edges) | O(|V| + |E|)(vertices + edges) |
| Space Complexity | O(|V|) (queue storage) | O(h) (max recursion depth), worst case O(|V|) |
| Completeness | Complete (finds solution if it exists) | Not complete (may get stuck in infinite loops without visited set) |
| Optimality | Optimal for unweighted graphs (shortest path) | Not optimal (may find a longer path to the goal) |

|  |  |  |
| --- | --- | --- |
| When to Use | Shortest path in unweighted graphs, web crawlers, social network analysis | Topological sorting, finding connected components, solving puzzles (e.g., mazes) |

Key Points:

- BFS is preferred when you need the shortest path in an unweighted graph or when exploring nodes level by level.

- DFS is useful for searching graphs or trees where the solution is deep or when memory is a concern (due to its lower space complexity in some cases).

Let me know if you'd like further clarification on any of these points!

|  |  |  |  |
| --- | --- | --- | --- |
| **Step** | **Queue** | **Visited Node** | **Level** |
| 1 | [A] | A | 0 |
| 3 | [C, D, E] | C | 1 |
| 4 | [D, E, F, G] | D | 2 |
| 5 | [E, F, G] | E | 2 |
| 6 | [F, G] | F | 2 |
| 7 | [G] | G | 2 |
| 8 | [] |  |

For DFS:

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

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **Step** | **Stack** | **Visited Node** |
| 1 | [A] | A |
| 2 | [C, B] | B |
| 3 | [C, E, D] | D |
| 4 | [C, E] | E |
| 5 | [C] | C |
| 6 | [G, F] | F |
| 7 | [G] | G |
| 8 | [] | — |