**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?

Maze

* **Initial State**: Position of the agent (e.g., top-left corner).
* **Possible Actions**: Move up, down, left, or right if not blocked by walls.
* **Goal Test**: Agent reaches the exit point (e.g., bottom-right corner).
* **Path Cost**: Number of moves or distance traveled.

City Map (Navigation from A to B)

* **Initial State**: Location of the traveler (e.g., City A).
* **Possible Actions**: Travel to directly connected cities via roads.
* **Goal Test**: Arrival at destination city (e.g., City B).
* **Path Cost**: Distance, time, or fuel cost—based on problem context.

Robot Vacuum Cleaner

* **Initial State**: Robot’s current location and which tiles are clean/dirty.
* **Possible Actions**:
  + Move in any direction (if within bounds).
  + Clean the current tile.
* **Goal Test**: All tiles are clean.
* **Path Cost**: Time, number of actions, or battery usage.

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

Fill in a table comparing:

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

| **Criteria** | **BFS (Breadth-First Search)** | **DFS (Depth-First Search)** |
| --- | --- | --- |
| **Data Structure Used** | Queue | Stack (or Recursion) |
| **Time Complexity** | O(V + E) (V = vertices, E = edges) | O(V + E) |
| **Space Complexity** | O(V) (stores all nodes at current level) | O(H), where H = height of tree or depth of graph |
| **Completeness** | Yes (if the branching factor is finite) | No (can get stuck in loops or dead ends if not handled) |
| **Optimality** | Yes (if all edge costs are equal) | No (returns first solution it finds, may not be optimal) |
| **When to Use** | - When shortest path is needed - For shallow solutions | - When solution is deep - Memory is limited |

**🔹 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

A

/ \

B C

/ \ / \

D E F G

#### ****Visited Nodes at Each Level:****

**Level 0**: A

**Level 1**: B, C

**Level 2**: D, E, F, G

#### ****BFS Queue at Each Step:****

| **Step** | **Queue Contents** | **Visited Node** |
| --- | --- | --- |
| 1 | A | - |
| 2 | B, C | A |
| 3 | C, D, E | B |
| 4 | D, E, F, G | C |
| 5 | E, F, G | D |
| 6 | F, G | E |
| 7 | G | F |
| 8 | (empty) | G |
|  |  |  |

### ****Depth-First Search (DFS)****

(Using **pre-order**: Root → Left → Right)

#### ****Order of Node Visits:****

mathematica

CopyEdit

A → B → D → E → C → F → G

#### ****DFS Stack at Each Step (Manual Traversal Simulation):****

| **Step** | **Stack (Top on Right)** | **Visited Node** |
| --- | --- | --- |
| 1 | A | - |
| 2 | C, B | A |
| 3 | C, E, D | B |
| 4 | C, E | D |
| 5 | C | E |
| 6 | G, F | C |
| 7 | G | F |
| 8 | (empty) | 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

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F', 'G'],

'D': [],

'E': [],

'F': [],

'G': []

}

BFS (Breadth-First Search)

from collections import deque

def bfs(graph, start, goal):

visited = []

queue = deque([start])

while queue:

print("Queue:", list(queue))

node = queue.popleft()

if node not in visited:

visited.append(node)

print("Visited:", visited)

if node == goal:

print(f"Goal node '{goal}' found!")

return

queue.extend([n for n in graph[node] if n not in visited])

print(f"Goal node '{goal}' not found.")

DFS (Depth-First Search)

def dfs(graph, start, goal):

visited = []

stack = [start]

while stack:

print("Stack:", stack)

node = stack.pop()

if node not in visited:

visited.append(node)

print("Visited:", visited)

if node == goal:

print(f"Goal node '{goal}' found!")

return

stack.extend(reversed([n for n in graph[node] if n not in visited]))

print(f"Goal node '{goal}' not found.")

**🔹 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

**Depth-Limited DFS (DLS)**

**def dls(graph, start, goal, limit):**

**visited = []**

**stack = [(start, 0)] # Stack holds tuples: (node, depth)**

**while stack:**

**print("Stack:", stack)**

**node, depth = stack.pop()**

**if node not in visited:**

**visited.append(node)**

**print("Visited:", visited)**

**if node == goal:**

**print(f"Goal node '{goal}' found at depth {depth}!")**

**return**

**if depth < limit:**

**# Push children with incremented depth**

**for neighbor in reversed(graph[node]):**

**if neighbor not in visited:**

**stack.append((neighbor, depth + 1))**

**print(f"Goal node '{goal}' not found within depth limit {limit}.")**

### ****Test Graph****

python

CopyEdit

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F', 'G'],

'D': [],

'E': [],

'F': [],

'G': []

}

**🧠 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

Maze Representation

# 0 = free cell, 1 = wall

maze = [

[0, 1, 0, 0, 0],

[0, 1, 0, 1, 0],

[0, 0, 0, 1, 0],

[1, 1, 0, 1, 0],

[0, 0, 0, 0, 0]

]

start = (0, 0)

goal = (4, 4)

### Helper: Print Maze with Path

python

CopyEdit

def print\_maze\_with\_path(maze, path):

maze\_copy = [['█' if cell == 1 else '.' for cell in row] for row in maze]

for x, y in path:

if (x, y) != start and (x, y) != goal:

maze\_copy[x][y] = '\*'

maze\_copy[start[0]][start[1]] = 'S'

maze\_copy[goal[0]][goal[1]] = 'G'

for row in maze\_copy:

print(' '.join(row))

BFS for Shortest Path

from collections import deque

def bfs\_maze(maze, start, goal):

n = len(maze)

m = len(maze[0])

visited = set()

queue = deque([(start, [start])]) # (current\_pos, path)

while queue:

(x, y), path = queue.popleft()

if (x, y) == goal:

print("\nBFS (Shortest Path) Found:")

print\_maze\_with\_path(maze, path)

return path

for dx, dy in [(-1,0), (1,0), (0,-1), (0,1)]:

nx, ny = x + dx, y + dy

if 0 <= nx < n and 0 <= ny < m and maze[nx][ny] == 0 and (nx, ny) not in visited:

visited.add((nx, ny))

queue.append(((nx, ny), path + [(nx, ny)]))

print("No path found using BFS.")

return None

### DFS for Any Path

python

CopyEdit

def dfs\_maze(maze, start, goal):

n = len(maze)

m = len(maze[0])

visited = set()

stack = [(start, [start])]

while stack:

(x, y), path = stack.pop()

if (x, y) == goal:

print("\nDFS (Any Path) Found:")

print\_maze\_with\_path(maze, path)

return path

if (x, y) not in visited:

visited.add((x, y))

for dx, dy in [(1,0), (-1,0), (0,1), (0,-1)]:

nx, ny = x + dx, y + dy

if 0 <= nx < n and 0 <= ny < m and maze[nx][ny] == 0:

stack.append(((nx, ny), path + [(nx, ny)]))

print("No path found using DFS.")

return None

### 🔹 Run Both Solvers

python

CopyEdit

bfs\_maze(maze, start, goal)

dfs\_maze(maze, start, goal)

**🔹 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

### ****1. State Representation****:

A state can be represented as a tuple:

python

CopyEdit

State = (x, y, energy\_left)

### ****2. Action Model****:

The drone can take 4 basic actions:

python

CopyEdit

Actions = [UP, DOWN, LEFT, RIGHT]

### ****2. Action Model****:

The drone can take 4 basic actions:

python

CopyEdit

Actions = [UP, DOWN, LEFT, RIGHT]

### ****4. Search Strategy Used & Why****:

#### 🔍 ****Strategy: A\* Search****

**Why A\***

**Efficiency**: Finds the **optimal path** while avoiding unnecessary exploration.

Uses a **heuristic**: Manhattan distance to goal (abs(x1 - x2) + abs(y1 - y2)), which works well in grid-based maps.

Accounts for variable **energy cost** (realistic terrain modeling).

### ****Bonus Enhancements (Optional for AI Simulation)****:

**Dynamic weather** updates affecting some grid cells

**Time constraints** on delivery (to add urgency)

**Multiple deliveries** with route optimization