Batch: B – 1 Roll no.: 16014022050

Experiment No.: 3

Title: Implementation of uninformed search algorithms – BFS, DFS, UCS for the given problem.

Course Outcome:

CO2: Describe & implement AI Search and optimization techniques.

Books/ Journals/ Websites referred:

1. **"Artificial Intelligence: a Modern Approach" by Russell and Norving, Pearson education Publications**
2. **"Artificial Intelligence" By Rich and knight, Tata McGraw Hill Publications**

Introduction:

Breadth-First Search (BFS)

Concept:

BFS is a graph traversal algorithm that explores nodes in **layers**. Starting from the root (or starting node), it visits all the nodes at distance `d` before visiting any nodes at distance `d+1`. This ensures that the first time you reach a node, you have found the shortest path to it (in terms of number of edges), provided that all edge costs are equal.

How it works:

1. Start from the initial node, mark it as visited, and place it in a **queue**.
2. Repeatedly remove the front node from the queue.
3. Visit all unvisited neighbors of that node and add them to the back of the queue.
4. Continue until either the goal is found or the queue is empty.

Characteristics:

- **Data Structure:** Queue (FIFO).
- **Completeness:** Yes, guaranteed to find a solution if one exists.
- **Optimality:** Yes, for equal edge costs.
- **Time Complexity:** $O(b^d)$, where:
  - b = branching factor (max number of children per node)
  - d = depth of the shallowest goal node.
- **Space Complexity:** $O(b^d)$ can be high due to storing all nodes at a given level.

Depth-First Search (DFS)

Concept:

DFS dives **deep** into the graph before backtracking. It follows one path until it can't go further, then backtracks to explore alternatives. It is typically implemented **recursively** or using a stack.

How it works:

1. Start at the root node, mark it visited.
2. Visit the first unvisited neighbor, then the next, going as far as possible down one branch.
3. When no further neighbors are found, backtrack to the most recent node with unvisited neighbors and repeat.

Characteristics:

- **Data Structure:** Stack (LIFO) or recursion.
- **Completeness:** No, can get stuck in cycles/infinite paths in infinite graphs without cycle checking.
- **Optimality:** No, may find a longer path before the shortest.
- **Time Complexity:** $O(b^m)$, where m = maximum depth.
- **Space Complexity:** $O(b^m)$ much less than BFS for large m.

Depth-Limited Search (DLS)

Concept:

DLS is DFS with a **maximum depth limit** L. It avoids the risk of infinite descent in cyclic or infinite graphs by cutting off the search beyond L levels.

How it works:

1. Perform DFS but keep track of the current depth.
2. If the depth limit is reached, stop exploring that branch.
3. Continue searching other branches within the limit.

When to use:

- In large/infinite graphs where BFS is too memory-heavy and DFS risks infinite loops.
- As a building block for **Iterative Deepening Search**.

Characteristics:

- **Completeness:** Yes, if $L \geq$ depth of the goal.
- **Optimality:** No, unless L is exactly the shallowest goal depth.
- **Time Complexity:** $O(b^L)$
- **Space Complexity:** $O(bL)$

Uniform Cost Search (UCS)

Concept:

UCS is a **cost-based** search strategy. Instead of expanding nodes in order of depth (like BFS), it expands nodes in order of **path cost** g(n)g(n)g(n) from the start node.

How it works:

1. Maintain a **priority queue** (min-heap) ordered by cumulative path cost.
2. Always remove and expand the node with the **lowest cost**.
3. Add/update neighbors in the priority queue with their total path cost.
4. Stop when the goal node is removed from the queue.

Characteristics:

- **Data Structure:** Priority Queue (Min-Heap).
- **Completeness:** Yes (if step costs ≥ 0).
- **Optimality:** Yes, always finds the cheapest solution.
- **Time Complexity:** $O(b^{1+[C^*/\epsilon]})$, where:
  - C^* = cost of optimal solution
  - ϵ = smallest positive step cost.
- **Space Complexity:** Can be high, needs to store all frontier nodes.

8 Puzzle Problem

Concept:

The 8 Puzzle is a classic problem in AI search:

- **Board:** 3×3 grid with 8 tiles numbered 1–8 and 1 empty space (blank).
- **Goal:** Rearrange tiles from a given initial configuration to the goal configuration by sliding tiles into the blank space.

Why it's interesting:

- It has **state space** of 9!/2=181,440 solvable states.
- It's a good demonstration of search algorithms, heuristics, and optimal path finding.

State Representation:

- As a list or tuple of 9 integers, with `0` representing the blank.

Operators:

- Move blank **up, down, left, right** (when possible).

Algorithm Choice:

- BFS → Finds shortest moves (but memory-heavy).
- DFS → May find a long or non-optimal solution.
- UCS / A* → Finds optimal moves considering move cost.

Implementation:

1. **BFS, DFS, DLS –**

```
from collections import deque

# ----------- BFS -----------
def bfs(graph, start):
    visited = set()
    queue = deque([start])

    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            queue.extend(graph.get(node, []))

# ----------- DFS -----------
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    if start not in visited:
        print(start, end=" ")
        visited.add(start)
        for neighbor in graph.get(start, []):
            dfs(graph, neighbor, visited)

# ----------- DLS -----------
def dls(graph, start, limit, depth=0, visited=None):
    if visited is None:
        visited = set()
    if start not in visited:
        print(start, end=" ")
        visited.add(start)
        if depth < limit:
            for neighbor in graph.get(start, []):
                dls(graph, neighbor, limit, depth + 1, visited)

# ----------- MAIN PROGRAM -----------
if __name__ == "__main__":
    # Example graph (Adjacency List)
    graph = {
        'A': ['B', 'C'],
        'B': ['D', 'E'],
        'C': ['F'],
        'D': [],
        'E': [],
```

```
        'F': []
    }
    start = 'A'

    print("BFS Traversal:")
    bfs(graph, start)
    print("\nDFS Traversal:")
    dfs(graph, start)
    print("\nDLS Traversal (limit = 2):")
    dls(graph, start, limit=2)
```

```
PS C:\Users\admin\OneDrive\Desktop\me\DSA\strings> python -u "c:\Users\
admin\OneDrive\Desktop\me\DSA\strings\x.py"
BFS Traversal:
A B C D E F
DFS Traversal:
A B D E C F
DLS Traversal (limit = 2):
A B D E C F
PS C:\Users\admin\OneDrive\Desktop\me\DSA\strings>
```

2. UCS –

```
import heapq

def ucs(graph, start, goal):
    pq = [(0, start)]  # (cost, node)
    visited = set()

    while pq:
        cost, node = heapq.heappop(pq)
        if node not in visited:
            print(f"Visiting {node} with total cost {cost}")
            visited.add(node)
            if node == goal:
                print(f"Goal reached! Total cost = {cost}")
                return
            for neighbor, weight in graph.get(node, []):
                heapq.heappush(pq, (cost + weight, neighbor))

if __name__ == "__main__":
    # Example weighted graph
    graph = {
        'A': [('B', 1), ('C', 5)],
        'B': [('D', 2), ('E', 4)],
        'C': [('F', 1)],
```

```
        'D': [],
        'E': [],
        'F': []
    }

    start = 'A'
    goal = 'F'
    print("Uniform Cost Search Traversal:")
    ucs(graph, start, goal)
```

```
PS C:\Users\admin\OneDrive\Desktop\me\DSA\strings> python -u "c:\Users\
admin\OneDrive\Desktop\me\DSA\strings\x.py"
Uniform Cost Search Traversal:
Visiting A with total cost 0
Visiting B with total cost 1
Visiting D with total cost 3
Visiting C with total cost 5
Visiting E with total cost 5
Visiting F with total cost 6
Goal reached! Total cost = 6
PS C:\Users\admin\OneDrive\Desktop\me\DSA\strings>
```

3. 8 Puzzle –

```
from collections import deque

# Function to generate neighboring states
def get_neighbors(state):
    neighbors = []
    idx = state.index(0)  # 0 = blank space
    moves = []
    if idx not in [0, 1, 2]: moves.append(-3)  # Up
    if idx not in [6, 7, 8]: moves.append(3)   # Down
    if idx not in [0, 3, 6]: moves.append(-1)  # Left
    if idx not in [2, 5, 8]: moves.append(1)   # Right

    for move in moves:
        new_state = list(state)
        new_state[idx], new_state[idx + move] = new_state[idx + move],
new_state[idx]
        neighbors.append(tuple(new_state))
    return neighbors

# BFS to solve 8 puzzle
def bfs_8_puzzle(start, goal):
    queue = deque([(start, [])])
```

```
    visited = set([start])

    while queue:
        state, path = queue.popleft()
        if state == goal:
            print("Solution found in", len(path), "moves:")
            for step in path:
                print(step)
            return
        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path + [neighbor]))

if __name__ == "__main__":
    # Example start and goal states
    start_state = (1, 2, 3,
                   4, 0, 5,
                   6, 7, 8)  # 0 = blank

    goal_state = (1, 2, 3,
                  4, 5, 6,
                  7, 8, 0)

    print("Solving 8 Puzzle Problem using BFS...")
    bfs_8_puzzle(start_state, goal_state)
```

```
PS C:\Users\admin\OneDrive\Desktop\me\DSA\strings> python -u "c:\Users\
admin\OneDrive\Desktop\me\DSA\strings\x.py"
Solving 8 Puzzle Problem using BFS...
Solution found in 14 moves:
(1, 2, 3, 4, 5, 0, 6, 7, 8)
(1, 2, 3, 4, 5, 8, 6, 7, 0)
(1, 2, 3, 4, 5, 8, 6, 0, 7)
(1, 2, 3, 4, 5, 8, 0, 6, 7)
(1, 2, 3, 0, 5, 8, 4, 6, 7)
(1, 2, 3, 5, 0, 8, 4, 6, 7)
(1, 2, 3, 5, 6, 8, 4, 0, 7)
(1, 2, 3, 5, 6, 8, 4, 7, 0)
(1, 2, 3, 5, 6, 0, 4, 7, 8)
(1, 2, 3, 5, 0, 6, 4, 7, 8)
(1, 2, 3, 0, 5, 6, 4, 7, 8)
(1, 2, 3, 4, 5, 6, 0, 7, 8)
(1, 2, 3, 4, 5, 6, 7, 0, 8)
(1, 2, 3, 4, 5, 6, 7, 8, 0)
PS C:\Users\admin\OneDrive\Desktop\me\DSA\strings>
```

Post Lab Descriptive Questions:

1. **What is the fundamental difference between BFS and DFS in terms of traversal strategy?**
   BFS (Breadth-First Search) explores nodes level by level — it visits all neighbors of a node before moving to the next level. It uses a queue (FIFO) to store nodes. This makes BFS suitable for finding the shortest path in unweighted graphs.
   DFS (Depth-First Search) explores as far as possible down a branch before backtracking. It uses a stack (LIFO) or recursion. DFS is memory-efficient but may go deep into a wrong path and miss shorter solutions.

2. **Give an example of a graph where DFS would fail to find the optimal solution, but UCS would succeed.**
   A --(1)--> B --(50)--> Goal
   A --(2)--> C --(2)--> D --(2)--> Goal

   DFS might explore A → B → Goal first (cost = 51) and stop there, missing the shorter path.
   UCS would explore paths in order of lowest cumulative cost, so it would choose A → C → D → Goal (cost = 6), which is optimal.

3. **Describe a real-world problem where UCS would outperform BFS and DFS.**
   Example: Navigation systems (Google Maps, GPS routing)
   - Roads have different travel times due to distances, speed limits, and traffic conditions.
   - BFS treats all roads equally and would not necessarily give the fastest route.
   - DFS might take a long detour before finding the goal.
   - UCS will always expand the path with the least travel time so far, guaranteeing the quickest route is found first.