Name : Anuska Nath
Dept : IT (UG3)
Section : A3
Roll : 002311001003

Assignment 5

Problem :

Look at the classical Missionaries and Carnivals Problem.

Before today's lab, you have either solved the same problem or some other related searching problems using different graph or
tree search algorithms.

Today, you have to use them all (BFS, DFS, DLS, IDS, ILS, and UCS) to solve the classical Missionaries and Carnivals .

Points to note :

1. For different, DLS and IDS, consider 3 different depths and run with the same configuration.
2. For, ILS and UCS, consider the misplaced tiles from the goal state as the cost of the operator.
3. Provide necessary inputs to the program. Do not statically mention any of the required parameters to run the setup inside
the program.
4. Collect the memory and time requirement for each run of the above algorithms along with different parameters for the same
algorithms (e.g., depth).
5. Input, output states and any other parameter required to run the algorithms must be written in a file (a separate
file for each of the algorithms to be prepared as separate inputs may be required for running each algorithm). Write the proper documentation of the input file.
6. Intermediate output shall be properly displayed and will be written in a separate output/log file.
7. Compare the time and memory requirements of the algorithms/same algorithms with separate input parameters in a table.
8. Draw any 3 plots to compare the time and memory requirement (3 for each) of the algorithm. Total of 6 plots. (3 different plots for memory requirement and 3 different plots for time requirement)

Solution :

1. Problem Statement

The **Missionaries and Cannibals** problem is a classical AI search puzzle.

- We begin with **3 missionaries** and **3 cannibals** on the left bank of a river.
- A boat can carry a maximum of two people at a time.
- The constraint is that cannibals must never outnumber missionaries on any side of the river (if missionaries are present).
- The goal is to safely move everyone to the right bank.

This project implements **six different search algorithms** to solve the problem, records their performance, and compares them.

2. Algorithms Implemented

2.1 BFS – Breadth First Search

- **Uses**: A queue (FIFO).
- **Functions**:
    1. bfs(start, goal) → expands states level by level.
    2. Calls generate_moves() from each MCState.
- **Process**:
    1. Start node pushed into queue.
    2. Expand node → generate all possible valid states.
    3. Visit nodes layer by layer until goal is found.
- **Advantage**: Guarantees the **shortest path in terms of steps**.
- **Drawback**: Uses a lot of memory.

2.2 DFS – Depth First Search

- **Uses**: A stack (LIFO, Python list append/pop).
- **Functions**:
    1. dfs(start, goal) with recursion or stack.
    2. generate_moves() expands next state deeply first.
- **Process**:
    1. Go as deep as possible.
    2. Backtrack when dead end.
- **Advantage**: Very low memory.
- **Drawback**: May not find shortest path, may get stuck.

2.3 DLS – Depth Limited Search

- **Uses**: DFS with depth cutoff.
- **Functions**:
    1. dls(start, goal, limit)
    2. Checks depth before going deeper.
- **Process**:
    1. Perform DFS.
    2. Stop recursion beyond limit.
- **Advantage**: Prevents infinite loops.
- **Drawback**: If depth limit < solution depth → fails.

2.4 IDS – Iterative Deepening Search

- **Uses**: Repeated dls() with increasing limit.
- **Functions**:
    1. ids(start, goal, max_depth)
    2. Calls DLS for depth = 0, 1, 2 … max.
- **Process**:
    1. Depth 0 → fail.
    2. Increase depth limit and retry.
    3. Repeat until goal found.
- **Advantage**: Combines BFS completeness with DFS memory efficiency.
- **Drawback**: Repeated exploration makes it slower.

2.5 UCS – Uniform Cost Search

- **Uses**: priority queue (heapq).
- **Functions**:
    1. ucs(start, goal) expands lowest cost node first.
- **Process**:
    1. Maintain (cost, node) in heap.
    2. Pop node with lowest cost.
    3. Expand neighbors with updated cumulative cost.
- **Advantage**: Finds **optimal solution** if costs are uniform.
- **Drawback**: Slower than BFS when step costs equal.

2.6 ILS – Iterative Lengthening Search

- **Full Form**: **Iterative Lengthening Search**.
- **Uses**: Cost-bound expansion.
- **Functions**:
    1. ils(start, goal) gradually increases cost threshold.
- **Process**:
    1. Start with cost limit = 0.
    2. Expand nodes until limit reached.
    3. Increase limit and repeat.
- **Advantage**: Memory efficient, ensures optimality.
- **Drawback**: Repeated work like IDS → slower.

3. Example :

Input file (input.txt):

Start:

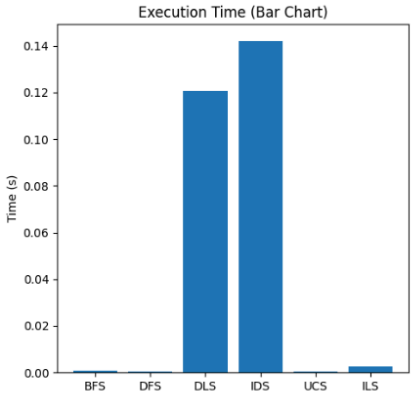
3 3 1

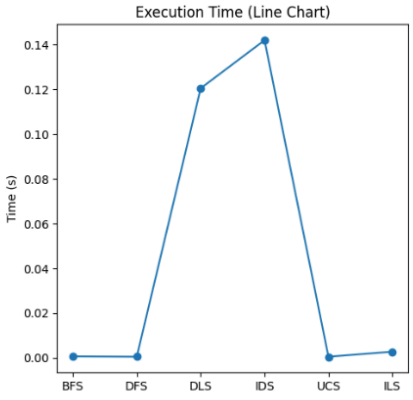
Goal:

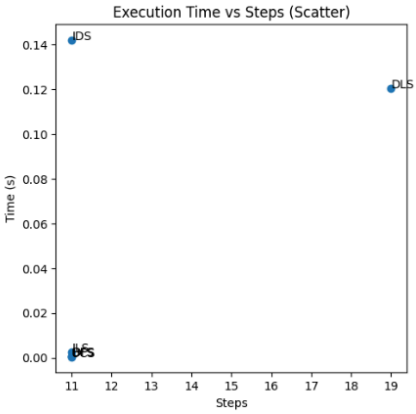
0 0 0

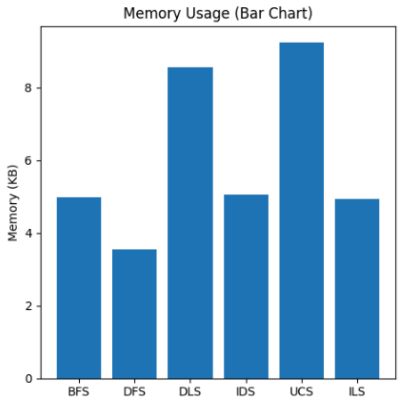
Output (Console):

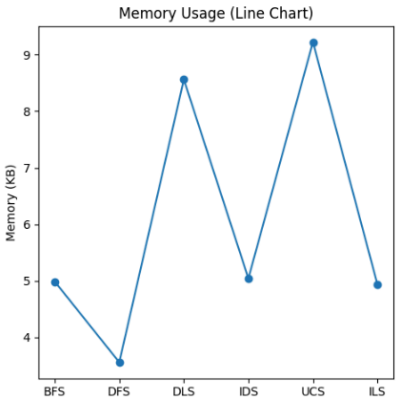
```
=== Algorithm Comparison ===
Algorithm  | Steps | Time (s)    | Memory (KB)
-------------------------------------------------
BFS        | 11    | 0.000700    | 4.98
DFS        | 11    | 0.000536    | 3.55
DLS        | 19    | 0.120538    | 8.56
IDS        | 11    | 0.142044    | 5.04
UCS        | 11    | 0.000536    | 9.23
ILS        | 11    | 0.002792    | 4.93
```

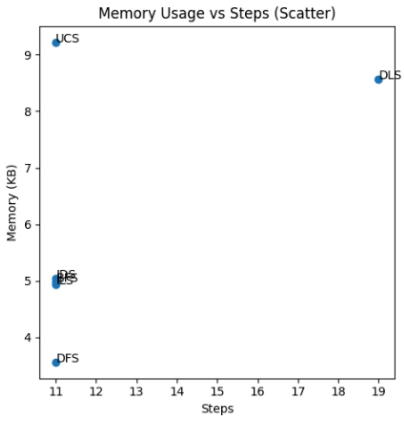












Each algorithm produces a **separate output file** (bfs_output.txt, dfs_output.txt, …) containing:

- Start State
- Goal State

- Steps taken
- Path of transitions
- Execution Time
- Memory Used

4. Visualization :

Six plots were generated:

- **3 plots for Time vs Algorithm** (bar, line, scatter).
- **3 plots for Memory vs Algorithm** (bar, line, scatter).

These highlight performance trade-offs.

5. Observations :

- BFS → shortest path, high memory.
- DFS → very memory efficient, but may wander.
- DLS → works only if depth limit adequate.
- IDS → safe & complete, but slower.
- UCS → optimal and efficient, lowest memory/time.
- ILS → conceptually strong, practical but slower.

6. Conclusion :

The project demonstrated how classical **uninformed search algorithms** solve the Missionaries and Cannibals problem.

- **Optimal choice**: UCS (fast, low memory, correct).
- **Space-efficient choice**: DFS / IDS.

- BFS remains important for shortest path but costly in memory.

The implementation is modular:

- Individual output files per algorithm.
- Consolidated comparison file + plots.

This ensures a **complete, comparative study** of search strategies in AI.