Experiment 1: Modeling and Solving Classical Search Problems Using State Space Search Techniques

Tejas Pakhle, Rajat Kumar Thakur, Tanay Patel, Abhi Tundya

***Abstract*—This report explores the use of state space search techniques to model and solve two classic problems: Missionaries and Cannibals and The Rabbit Leap. Both problems are framed as state space graphs, where Breadth-First Search and Depth-First Search are applied to find solutions. BFS explores all possible states level by level, ensuring optimal solution at the cost of memory consumption. In contrast, DFS dives deep into each possible state path, using less memory and with the risk of missing the optimal solution. By analyzing the performance of both techniques, the report highlights their respective strengths and weaknesses in terms of time and space complexity when applied to these specific problem types.**

I. Introduction

I tate space search techniques are particularly valuable for addressing problems, where the objective is to navigate through different configurations or states to arrive at a solution. This report examines two such problems—the Missionaries and Cannibals and the Rabbit Leap—which have been widely used as benchmarks in AI research.

The Missionaries and Cannibals involves a group of missionaries and cannibals who must be safely transported across a river using a boat, with the constraint that at no point should the cannibals outnumber the missionaries on either side.

The Rabbit Leap problem presents a different type of challenge, involving two groups of rabbits—one moving eastward and the other westward—who must cross each other along a sequence of stones placed in a straight line across a stream. The puzzle requires the rabbits to move forward or leap over one another while following the rules of the game. The challenge is to determine whether the rabbits can cross each other without violating the movement constraints or falling into the water.

Both problems serve as ideal case studies for the application of search techniques like Breadth-First Search (BFS) and Depth-First Search (DFS), allowing for a detailed analysis of search efficiency, optimality, and computational complexity in solving such puzzles.

II. UNDERSTANDING

A. *Search Algorithms*

Search algorithms provide systematic approaches to explore a state space, navigating through possible configurations to reach a solution. They are vital in artificial intelligence (AI) for addressing complex problems by transitioning from an initial state to a goal state based on predefined rules and constraints. Before diving into practical applications, it is important to understand the types of search algorithms, such as *Breadth-First Search (BFS)* and *Depth-First Search (DFS)*, and the distinction between *uninformed* and *informed* search strategies, including the role of heuristics in the latter.

B. *State Space*

A state space is the set of all possible configurations or arrangements that a problem can take. It forms the foundation for search techniques, as each state represents a unique point in the problem's progression.

Researchers must first define the state space carefully for problems like *Missionaries and Cannibals* or *Rabbit Leap*, outlining each possible combination of missionaries, cannibals, boats, or rabbits that could occur throughout the puzzle.

C. *Search Efficiency*

Search efficiency refers to how effectively a search algorithm can find the solution, typically measured in terms of time complexity (how long the algorithm takes) and space complexity (the amount of memory it uses).

Analyzing search efficiency involves comparing algorithms like BFS and DFS, which differ in how they explore the state space. Factors such as the depth of the solution and the branching factor of the problem significantly impact efficiency.

D. *Optimality*

Optimality ensures that a search algorithm finds not just any solution, but the best possible one. For instance, in puzzles like the *Rabbit Leap* problem, an optimal solution would be the one that requires the least moves or satisfies the goal with minimal cost.

The choice of the search alogrithm is largely dependent on the constraint of the problem as well as the available system resources. Researchers focus on search algorithms like BFS, which guarantees an optimal solution in terms of the shortest path, compared to DFS, which may find a solution but not necessarily the optimal one while they may also opt for DFS when time is a constraint, especially if the solution obtained is close to the optimal one and the difference is not significant. This approach is practical when the speed of finding a solution outweighs the need for exact optimality.

III. Problem Statement

A. Missionaries and Cannibals

The missionaries and cannibals problem is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place. This problem is famous in AI because it was the subject of the first paper that approached problem-formulation from an analytical viewpoint.

B. The Rabbit Leap

In the rabbit leap problem, three east-bound rabbits stand in a line blocked by three west-bound rabbits. They are crossing a stream with stones placed in the east west direction in a line. There is one empty stone between them. The rabbits can only move forward one or two steps. They can jump over one rabbit if the need arises, but not more than that. Are they smart enough to cross each other without having to step into the water?

C. Problem I

1) Model the problem as a state space search problem. How large is the search space?
2) Solve the problem using BFS. The optimal solution is the one with the fewest number of steps. Is the solution that you have acquired an optimal one? The program should print out the solution by listing a sequence of steps needed to reach the goal state from the initial state.
3) Solve the problem using DFS. The program should print out the solution by listing a sequence of steps needed to reach the goal state from the initial state.
4) Compare solutions found from BFS and DFS. Comment on solutions. Also compare the time and space complexities of both.

IV. Missionaries and Cannibals Problem

A. Problem Formulation

We represent the state as a tuple (M, C, B) where:

- M: Number of missionaries on the left side
- C: Number of cannibals on the left side
- B: Position of the Boat(1: Left bank, 2: Right Bank)

Initial State: $(3, 3, 1)$

Goal State: $(0, 0, 0)$

For missionaries, we have 4 possibilities (0, 1, 2, or 3 missionaries)

For cannibals, we also have 4 possibilities (0, 1, 2, or 3 cannibals)

For the boat, we have 2 possibilities (0 or 1)

$$=> 4 \times 4 \times 2 = 32 \text{ possible states} \quad (1)$$

B. BFS Solution

Algorithm 1 BFS for Missionaries and Cannibals Problem

```
procedure BFS(start_state, goal_state)
    queue ← deque([(start_state, [])])
    visited ← ∅
    while queue is not empty do
        (state, path) ← queue.popleft()
        if state ∈ visited then
            continue
        end if
        nodes_visited ← nodes_visited + 1
        path ← path + [state]
        if state = goal_state then
            print nodes_visited
            return path
        end if
        for successor ∈ get_successors(state) do
            queue.append((successor, path))
        end for
    end while
    return None
end procedure
```

C. DFS Solution

Algorithm 2 Depth-First Search for Missionaries and Cannibals Problem

```
procedure DFS(start_state, goal_state)
    stack ← deque([(start_state, [])])
    visited ← ∅
    while stack is not empty do
        (state, path) ← stack.pop()
        if state ∈ visited then
            continue
        end if
        visited.add(state)
        path ← path + [state]
        if state = goal_state then
            return path
        end if
        for successor ∈ get_successors(state) do
            stack.append((successor, path))
        end for
    end while
    return None
end procedure
```

D. Complexity Comparison

Complexity Type	BFS	DFS
Time Complexity	$O(b^d)$	$O(b^d)$
Space Complexity	$O(b^d)$	$O(bd)$
Total visited	15	14

Table I: Complexity Comparison of BFS and DFS

where b is branching factor and d is depth of shallowest goal state

V. Rabbit Leap Problem

A. Problem Formulation

We represent the state as :-
$[E1, E2, E3, \text{Empty}, W1, W2, W3]$, where:

- $W1, W2, W3$: West-bound rabbits (denoted: +1)
- $E1, E2, E3$: East-bound rabbits (denoted: -1)
- mpty: Position of the empty stone (Denoted: 0)

Initial State: $[-1, -1, -1, 0, 1, 1, 1]$
Goal State: $[1, 1, 1, 0, -1, -1, -1]$
Total number of states = $\frac{7!}{3!4!} = 35$

B. BFS Solution

Algorithm 3 Breadth-First Search for Rabbit Leap Problem

```
procedure BFS(start_state, goal_state)
    queue ← deque([(start_state, [])])
    visited ← ∅
    while queue is not empty do
        (state, path) ← queue.popleft()
        if state ∈ visited then
            continue
        end if
        visited.add(state)
        path ← path + [state]
        if state = goal_state then
            return path
        end if
        for successor ∈ get_successors(state) do
            queue.append((successor, path))
        end for
    end while
    print nodes_visited
    return None
end procedure
```

C. DFS Solution

Algorithm 4 Depth-First Search for Rabbit Leap Problem

```
procedure DFS(start_state, goal_state)
    stack ← deque([(start_state, [])])
    visited ← ∅
    while stack is not empty do
        (state, path) ← stack.pop()
        if state ∈ visited then
            continue
        end if
        visited.add(state)
        path ← path + [state]
        if state = goal_state then
            return path
        end if
        for successor ∈ get_successors(state) do
            stack.append((successor, path))
        end for
    end while
    return None
end procedure
```

D. Complexity Comparison

Complexity Type	BFS	DFS
Time Complexity	$O(b^d)$	$O(b^d)$
Space Complexity	$O(b^d)$	$O(bd)$
Total Visited	136	106

Table II: Complexity Comparison of BFS and DFS

where b is branching factor and d is depth of shallowest goal state

VI. Analysis

The time complexity of the BFS algorithm is $O(b^d)$, where b is the branching factor (the average number of successors per state), and d is the depth of the shallowest goal state. In this case, the `get_successors` function generates all possible successors for a given state, contributing to the branching factor. The BFS function's while loop iterates through the queue of states until the goal is reached or all nodes are visited, resulting in a time complexity of $O(b^d)$. Similarly, the time complexity of the DFS algorithm is also $O(b^d)$, where b is the branching factor (number of possible moves from each state), and d is the depth of the search tree. In the Rabbit Leap Problem, the branching factor is 4 (1 step or 2 steps in either direction), while the depth varies depending on the specific instance of the problem.

Depth-First Search (DFS) is more space-efficient than Breadth-First Search (BFS) because it explores one path to its maximum depth before backtracking, only needing to store the nodes along the current path. This leads to a space complexity of $O(bd)$, where b is the branching factor and d is the maximum depth of the tree. In contrast, BFS explores all nodes at each depth level before proceeding, requiring it to store all nodes at a given depth, resulting in a space complexity of $O(b^d)$, where d is the depth of the shallowest solution. In large or wide search spaces, BFS can quickly consume exponentially more memory, making DFS far more space-efficient when memory resources are limited. For the Rabbit Leap problem, the state space is quite structured, meaning that DFS might find the goal faster than BFS due to its depth-first nature. Since BFS explores all nodes at the same depth level before moving deeper, it tends to visit more nodes, especially if the solution is located at a deeper level in the tree. DFS, on the other hand, can reach the solution sooner if it explores the right branches first.

VII. Conclusion

In conclusion, this laboratory highlights the comparative analysis of Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms in solving the Rabbit Leap Problem. While both algorithms exhibit a time complexity of $O(b^d)$, DFS demonstrates greater space efficiency due to its depth-focused approach. The choice between BFS and DFS ultimately depends on the specific characteristics of the problem, including the branching factor and depth of the search tree. In the given set of problem the DFS was superior compared to DFS in all aspect.