State Space Search Problem and BFS/ DFS

Divyansh Nahar
Computer Science and Engineering
202351035@iiitvadodara.ac.in

Jinendra Kumar Jain
Computer Science and Engineering
202351050@iiitvadodara.ac.in

Kuldeep Purbiya
Computer Science and Engineering
202351072@iiitvadodara.ac.in

***Abstract*—This paper applies state space search algorithms to solve the Rabbit Leap problem. The problem is modeled as a state space search task, with solutions found using Breadth-First Search (BFS) and Depth-First Search (DFS). BFS guarantees optimal solutions with the fewest steps, while DFS explores alternative, potentially non-optimal paths. The study compares solution quality, time, and space complexities for both algorithms. Results show that BFS consistently finds optimal solutions with higher memory costs, while DFS is more memory-efficient but may yield suboptimal solutions. This analysis highlights the trade-offs between BFS and DFS in solving the Rabbit Leap problem.**

[twocolumn]article amsmath amssymb graphicx listings cite

I. Introduction

State space search is a fundamental technique in artificial intelligence, enabling the resolution of problems by navigating through various configurations, or "states," to achieve a desired goal. One classic problem that exemplifies this approach is the Rabbit Leap problem, which serves as an effective benchmark for evaluating search strategies and their performance in algorithmic contexts [1].

The Rabbit Leap problem involves three east-bound rabbits and three west-bound rabbits that must cross each other by jumping over stones, with the restriction that each rabbit can only leap over one rabbit at a time.

This paper models the Rabbit Leap problem as a state space search challenge, employing two widely used algorithms: Breadth-First Search (BFS) and Depth-First Search (DFS). BFS guarantees the optimal solution—defined as the path with the fewest steps—while DFS offers a more memory-efficient approach, albeit often leading to longer or suboptimal solutions. By comparing these algorithms in terms of solution quality, time complexity, and space complexity, this study aims to highlight the trade-offs between BFS and DFS in solving the Rabbit Leap problem, providing valuable insights into their applicability in artificial intelligence.

II. Objectives

The primary objective of this research is to analyze and solve the Rabbit Leap problem using state space search algorithms. The study aims to achieve the following specific goals:

A. Rabbit Leap Problem

Code-BFS Code-DSF

- **Modeling the Problem:** Define the Rabbit Leap problem as a state space search problem, identifying the initial configuration of rabbits, the goal state where all rabbits have crossed the stream, and the permissible movements.
- **Search Space Analysis:** Evaluate the search space size by considering all possible arrangements of the rabbits on the stones, including the implications of jumps and movements.
- **Algorithm Implementation:** Develop algorithms using BFS and DFS to navigate the state space, finding a sequence of steps that allows all rabbits to cross without stepping into the water.
- **Solution Evaluation:** Compare the solutions produced by BFS and DFS, discussing the efficiency and effectiveness of each algorithm in reaching the goal state.

The `is_valid(state)` function verifies that the position of the empty space ('.') is within the bounds of the state string.

```
def is_valid(state):
    return 0 <= state.index('.') < len(state)
```

The `get_successors(state)` function generates successor states by moving the empty space ('.') one or two positions left or right. It checks for valid moves based on the presence of 'W' (west-bound) and 'E' (east-bound) rabbits and adds valid new states to the successors list.

```
def get_successors(state):
 successors = []
 index = state.index('.')
 moves = [1, 2]
  for move in moves:
     if index + move < len(state):
     if state[index + move] == 'W':
       new_state = list(state)
       new_state[index], new_state[index+
       move] = new_state[index + move],
       new_state[index]
       new_state = ''.join(new_state)
     if is_valid(new_state):
        successors.append(new_state)
  for move in moves:
     if index - move >= 0:
     if state[index - move] == 'E':
       new_state = list(state)
       new_state[index], new_state[index-
       move] = new_state[index - move],
       new_state[index]
       new_state = ''.join(new_state)
```

```
        if is_valid(new_state):
            successors.append(new_state)
    return successors
```

References

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed., Pearson, 2022.