Week-1: Rabbit Leap — BFS and DFS State-Space Search

Chetankumar Kamani (20251603005), Sakariya Devraj (20251603006), and Divyesh Dodiya (20251603007)

I. Problem Statement

In the Rabbit Leap problem, three east-bound rabbits stand in a line blocked by three west-bound rabbits. They are crossing a stream using stones placed in a straight east–west direction. There is one empty stone between them. Each rabbit can only move forward (in its facing direction) by one or two stones at a time. A rabbit may jump over exactly one rabbit if needed, but not over two. The goal is to determine whether the rabbits can successfully cross each other without stepping into the water.

II. State Space Search

State representation: Each state is represented as a string of length 7 containing three east-bound rabbits (>), three west-bound rabbits (<), and one empty position (_).
Initial state: >>>_<<<
Goal state: <<<_>>>

Valid Moves

- $>$ can move one step right into the empty position.
- $<$ can move one step left into the empty position.
- $>$ can jump right over one $<$ into the empty position.
- $<$ can jump left over one $>$ into the empty position.

Search Space Size

- The total number of possible states is:

$$\text{Total possible states} = \frac{7!}{3!\,3!\,1!} = \frac{5040}{36} = 140$$

- Reachable states under valid moves $= 72$

III. Breadth-First Search (BFS) Solution

Breadth-First Search explores level by level, so the first solution found is optimal (i.e., it uses the minimum number of moves).

A. Python Code (BFS)

```
from collections import deque

def next_states(state):
    s = list(state)
    i = s.index('_')
    # adjacent moves
    if i - 1 >= 0 and s[i - 1] == '>':
        t = s.copy(); t[i], t[i - 1] = t[i - 1], t
            [i]
        yield ('> moves 1 right', ''.join(t))
    if i + 1 < len(s) and s[i + 1] == '<':
        t = s.copy(); t[i], t[i + 1] = t[i + 1], t
            [i]
        yield ('< moves 1 left', ''.join(t))
    # jumps over exactly one opponent
    if i - 2 >= 0 and s[i - 2] == '>' and s[i - 1]
        == '<':
        t = s.copy(); t[i], t[i - 2] = t[i - 2], t
            [i]
        yield ('> jumps over <', ''.join(t))
    if i + 2 < len(s) and s[i + 2] == '<' and s[i
        + 1] == '>':
        t = s.copy(); t[i], t[i + 2] = t[i + 2], t
            [i]
        yield ('< jumps over >', ''.join(t))

def bfs(start=">>>_<<<", goal="<<<_>>>"):
    q = deque([start])
    parent = {start: None}
    action = {}
    seen = {start}
    while q:
        u = q.popleft()
        if u == goal:
            break
        for a, v in next_states(u):
            if v not in seen:
                seen.add(v)
                parent[v] = u
                action[v] = a
                q.append(v)
    if goal not in parent:
        return None
    # reconstruct path
    path, s = [], goal
    while s is not None:
        path.append(s)
        s = parent[s]
    path.reverse()
    steps = []
    for i in range(1, len(path)):
        steps.append((action[path[i]], path[i]))
    return steps

steps = bfs()
print("BFS found steps:", len(steps))
for move, state in steps:
    print(f"{move:>18} -> {state}")
```

B. Execution Screenshot (BFS)

IV. Depth-First Search (DFS) Solution

Depth-First Search explores one path fully before backtracking. It is not guaranteed to find the optimal solution, but it can still find a valid one.

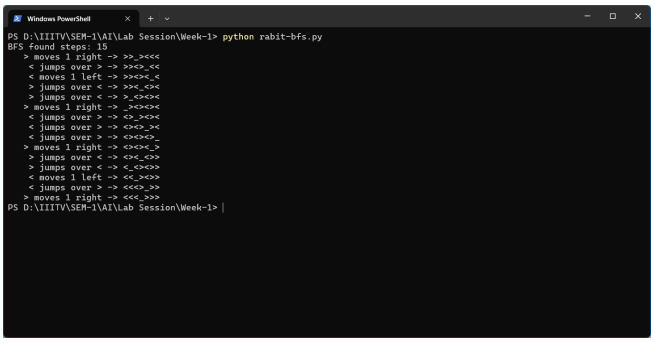

Fig. 1. BFS execution result.

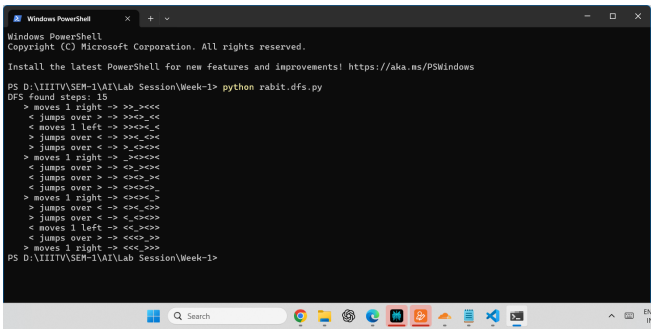

Fig. 2. DFS execution result.

A. Python Code (DFS)

```
def next_states(state):
    s = list(state)
    i = s.index('_')
    # adjacent moves
    if i - 1 >= 0 and s[i - 1] == '>':
        t = s.copy(); t[i], t[i - 1] = t[i - 1], t
            [i]
        yield ('> moves 1 right', ''.join(t))
    if i + 1 < len(s) and s[i + 1] == '<':
        t = s.copy(); t[i], t[i + 1] = t[i + 1], t
            [i]
        yield ('< moves 1 left', ''.join(t))
    # jumps over exactly one opponent
    if i - 2 >= 0 and s[i - 2] == '>' and s[i - 1]
       == '<':
        t = s.copy(); t[i], t[i - 2] = t[i - 2], t
            [i]
        yield ('> jumps over <', ''.join(t))
    if i + 2 < len(s) and s[i + 2] == '<' and s[i
        + 1] == '>':
        t = s.copy(); t[i], t[i + 2] = t[i + 2], t
            [i]
        yield ('< jumps over >', ''.join(t))

def dfs(start=">>>_<<<", goal="<<<_>>>"):
    stack = [(start, [])]
    visited = {start}
    while stack:
        state, path = stack.pop()
        if state == goal:
            return path
        for move, nxt in reversed(list(next_states
            (state))):
            if nxt not in visited:
                visited.add(nxt)
                stack.append((nxt, path + [(move,
                    nxt)]))
    return None

steps = dfs()
print("DFS found steps:", len(steps))
for move, state in steps:
    print(f"{move:>18} -> {state}")
```

B. Execution Screenshot (DFS)

V. Final Comparison

- **BFS:** Always finds the optimal (minimum number of moves) solution but required more memory.
- **DFS:** Required less memory but may not find the shortest path.

Aspect	BFS	DFS
Solution Type	Always optimal	Not guaranteed optimal
Time Complexity	$O(b^d)$	$O(b^m)$
Space Complexity	$O(b^d)$ (high)	$O(b \cdot m)$ (low)
Memory Usage	Large	Small

VI. Code Availability

The complete source code is available at: GitHub Repository (CS659 – AI Laboratory).

GitHub Repository:
https://github.com/ChetanKamani/CS659-AI-Lab