

Artificial Intelligence Lab Report 1

Anuj Saha
202351010
BTech CSE
IIIT, Vadodara

G. Nikhil
202351037
BTech CSE
IIIT, Vadodara

Divyanshu Ghosh
202351036
BTech CSE
IIIT, Vadodara

Abstract—The rabbit leap problem involves three east-bound rabbits and three west-bound rabbits positioned on a line of seven stones across a stream, with one empty stone separating them. The goal is to swap the positions to WWW_EEE, with rabbits moving only forward (east-bound to the right, west-bound to the left) by either stepping to an adjacent empty stone or jumping over one rabbit to an empty stone two steps away, without stepping into the water. This report models the problem as a state space search problem, solves it using Breadth-First Search (BFS) and Depth-First Search (DFS), and compares the solutions, including their time and space complexities. The analysis draws from the reference *Artificial Intelligence: A Modern Approach* by Russell and Norvig (Fourth Edition). Complete code implementation is available at: https://github.com/AnujSaha0111/CS307-Lab-Submissions/tree/main/Submission_1.

Index Terms—Rabbit leap, state space search, BFS, DFS, AI search, optimal solution

I. INTRODUCTION

The Rabbit Leap Problem is one of the archetypical puzzles seen in artificial intelligence, which involves three rabbits that move east (E) and three rabbits that move west (W) on the 7 stones spanning a stream, with a single empty stone separating the two types of rabbits. The starting position is represented by EEE_WWW, while the target (goal) position is WWW_EEE. All rabbits can only move directly forward: E obviously moves to the right, while W moves to the left. Each rabbit can either (1) step to an empty adjacent stone, or (2) jump over a single rabbit to the empty stone two spaces in front of the rabbit's current position.

This can be thought of as a state space search problem as described in Russell and Norvig's book *Artificial Intelligence: A Modern Approach* (4th Ed.; Chapters 1–3). The state space theoretically includes 140 possible configuration states (the confines of movement will obviously limit those actual reachable states, as described later). The report discusses each of the implementation as well as comparing the two solutions, BFS and DFS, and evaluates the performance of carefully examining the time complexity and space complexity of each.

II. PROBLEM MODELING

The Rabbit Leap Problem is formally defined by five components as per Section 3.1 of the reference:

- **States:** Configurations of seven positions with three 'E's, three 'W's, and one '_'. Example: [E, E, E, _, W, W, W].
- **Initial State:** [E, E, E, _, W, W, W].
- **Goal State:** [W, W, W, _, E, E, E].
- **Actions:** Move a rabbit to the empty stone if adjacent (step) or two positions away (jump over one rabbit), respecting direction constraints.

- **Transition Model:** Swap the moving rabbit with the empty stone.
- **Path Cost:** Each move costs 1 unit.

The search space size is calculated as follows:

- 7 positions for the empty stone: $\binom{7}{1} = 7$ ways
- 3 'E's and 3 'W's in remaining 6 positions: $\binom{6}{3} = 20$ ways
- Total: $7 \times 20 = 140$ states

However, directional constraints limit reachable states. The branching factor b ranges from 2–4 depending on the current configuration.

III. BFS IMPLEMENTATION

Breadth-First Search explores all nodes at the current depth before proceeding to the next level, guaranteeing the shortest path in unweighted graphs (Section 3.4.1, p. 94 in the reference).

Algorithm 1 BFS Graph Search Agent for Rabbit Leap Problem

Require: problem (initial state, goal test, successors function)
Ensure: path to goal or failure

```

1: Initialize empty queue, hash table for visited states
2: Enqueue initial state to queue
3: Add initial state to hash table
4: while queue is not empty do
5:   current_state ← Dequeue from queue
6:   if current_state is goal state then
7:     return success and path to goal
8:   end if
9:   for all successor_state in SUCCESSORS(current_state)
10:    do
11:      if successor_state not in hash table then
12:        Add successor_state to hash table
13:        Enqueue successor_state to queue
14:      end if
15:    end for
16: end while
17: return failure

```

Algorithm Explanation: The BFS algorithm executes a search of the graph using First-In-First-Out (FIFO) queue structure to search for states level by level through time. The algorithm begins with the initialization of an empty queue and a hash table to store visited states in order to avoid both cycles and repeated exploration of already visited states. The queue is initialized with the initial state and the initial state is marked

as visited. During each time step, the algorithm will dequeue the state at the front of the queue, if the goal state equals the current state, the algorithm will return the path to the goal state; else, the algorithm will execute the GET_SUCCESSEORS function in order to generate all the valid successor states, enqueueing the unvisited successor states while marking those other successor states as visited. The level by level search ensures that the first path found to the goal state will also be the shortest path through time; thus, the BFS is called optimal for unweighted graphs. The hash table provides average-case lookup time of $O(1)$, which is efficient for duplicate detection.

Algorithm 2 Generate Successors for Rabbit Leap Problem

Require: state (current configuration)

Ensure: list of successor states

- 1: Find position of empty stone (_)
 - 2: Initialize empty list of successors
 - 3: Generate possible moves:
 - 4: - Adjacent positions (one step forward)
 - 5: - Two positions away (jump over one rabbit)
 - 6: **for** each valid move respecting direction constraints **do**
 - 7: Create new state by swapping rabbit with empty stone
 - 8: Add new state to successors list
 - 9: **end for**
 - 10: **return** successors
-

Algorithm Explanation: The successor generation function is essential to the definition of state space exploration. It first finds the position of the empty stone (marker or pit) in the current configuration. It then finds all rabbits that can legally move to the empty position according to two rules: (1) A rabbit can step into an adjacent, empty stone (mark or pit) provided the rabbit is moving forward in the designated direction in which it can only move (E moves to the right, W moves to the left), and (2) A rabbit may jump over one rabbit to reach an empty stone that is distant two stones in the marked direction it moves, abstracting the possible jump if the stone is immediately next to or over the empty stone, but constrained to move in one direction. For each rabbit that can either slide to the empty stone or jump to the empty stone, the algorithm creates a new state by swapping the rabbit's current position with the empty stone (the same jump represented through the actions described previously) which mimics the jump and illustrates a possible move. The collection of all valid successor possible states is returned representing all the possibilities for the next state from the present state reachable through one move that is legally possible according to the rules specified. This function represents the application of the defined rules for this type of problem solving.

IV. DFS IMPLEMENTATION

Depth-First Search explores as far as possible along each branch before backtracking, which may not yield optimal solutions (Section 3.4.3, p. 96 in the reference).

TABLE I
BFS OPTIMAL SOLUTION (15 STEPS)

Step	Move	State
Initial	-	EEE_WWW
1	E at 2→3	EE_EWWW
2	W at 4→2	EEWE_WW
3	W at 5→4	EEWEW_W
4	E at 3→5	EEW_WEW
5	E at 1→3	E_WEWEW
6	E at 0→1	_EWEWEW
7	W at 2→0	WE_EWEW
8	W at 4→2	WEWE_EW
9	W at 6→4	WEWEWE_
10	E at 5→6	WEWEW_E
11	E at 3→5	WEW_WEE
12	E at 1→3	W_WEWEE
13	W at 2→1	WW_EWEE
14	W at 4→2	WWWE_EE
15	E at 3→4	WWW_EEE

Algorithm 3 DFS Graph Search Agent for Rabbit Leap Problem

Require: problem (initial state, goal test, successors function)

Ensure: path to goal or failure

- 1: Initialize stack, hash table for visited states
 - 2: Push initial state to stack
 - 3: Add initial state to hash table
 - 4: **while** stack is not empty **do**
 - 5: current_node ← Pop from stack
 - 6: **if** current_node.state is goal state **then**
 - 7: **return** BACKTRACK_PATH(current_node)
 - 8: **end if**
 - 9: **for** all successor_state **in** SUCCESSORS(current_state) **do**
 - 10: **if** successor_state not **in** hash table **then**
 - 11: Create new node with successor_state
 - 12: Set new node.parent = current_node
 - 13: Add successor_state to hash table
 - 14: Push new node to stack
 - 15: **end if**
 - 16: **end for**
 - 17: **end while**
 - 18: **return** failure
-

Algorithm Explanation: The DFS algorithm employs a Last-In-First-Out (LIFO) stack to explore states in depth-first manner. Unlike BFS, DFS dives deep into one branch of the search tree before backtracking. The algorithm initializes a stack and hash table, then it pushes the initial state onto the stack. In each iteration, it pops the top node from the stack and checks for goal state. If the state popped is not the goal, it generates all unvisited successor states and pushes them onto the stack as new nodes, each keeping the track of its parent for path reconstruction. This parent-child relationship is crucial because DFS doesn't naturally maintain path information like BFS does. The LIFO nature means the most recently discovered states are explored first, leading to deep exploration of branches. While memory-efficient (storing only the current path), DFS may find suboptimal solutions since it doesn't explore systematically by depth level. The hash table still prevents revisiting states, ensuring the algorithm terminates in finite state spaces.

TABLE II
DFS EXAMPLE SOLUTION (21 STEPS - NON-OPTIMAL)

Step	Move	State
Initial	-	EEE_WWW
1	E at 2→3	EE_EWWW
2	W at 4→2	EEWE_WW
3	E at 1→4	E_EWWEW
4	W at 5→1	EW_EWWE
5	W at 6→5	EW_EWW_
6	E at 0→6	_E_EWW_
7	W at 1→0	WE_EWW_
8	E at 3→1	WEE_WW_
9	W at 5→3	WEEW_W_
10	E at 1→5	WE_WWE_
11	W at 0→1	_WEWWE_
12	E at 4→0	EW_WWE_
13	W at 3→4	EW_WWE_
14	E at 0→3	_W_WWE_
15	W at 2→0	WW_W_E
16	E at 3→1	W_WW_E
17	E at 4→3	W_WE_W
18	W at 0→4	_WE_WW
19	E at 1→0	E_W_WW
20	W at 2→1	EW_W_W
21	E at 0→2	WWW_EEE

Algorithm 4 Backtrack Path from Goal Node

Require: node (goal node)

Ensure: path from initial state to goal

- 1: Initialize empty path
 - 2: **while** node is not null **do**
 - 3: Insert node.state at beginning of path
 - 4: node ← node.parent
 - 5: **end while**
 - 6: **return** path
-

Algorithm Explanation: The process of backtracking creates the solution path from the goal node to the original state, following parent pointers. When DFS finds the goal, the goal node is all that it has; it does not know the entire path. The backtracking function traverses the parent chain, and, for each node's state, it builds the path backwards by adding the state to the front of the path list and moving to the parent. This process repeats until the initial state is reached (which has no parent, hence null). The backtrack process is necessary because DFS explores depth-first, and the path from the initial state to the goal state is not maintained as it is for BFS. Parent pointers are an efficient way to represent path information. Since only a single branch requires stack space, we record only depth m instead of all of the previously explored paths, considering the nodes explored could number m . Ultimately, the parent pointer mechanism only takes $O(m)$ space in the stack.

An example DFS execution produced a 21-step solution:

V. COMPARISON OF BFS AND DFS SOLUTIONS

A. Solution Quality Analysis

BFS always gives the optimal solution of 15 steps because it explores all the states in levels based on distance from the start state to the goal adding all states at each level to the search space. This characteristic provides complete and optimal solutions in a finite state space.

In the example, DFS produced a solution that was 21 steps long, exhibiting a tendency to go to sub-optimal solution paths

TABLE III
TIME AND SPACE COMPLEXITY COMPARISON

Property	BFS	DFS
Time Complexity	$O(b^{d+1})$	$O(b^m)$
Space Complexity	$O(b^{d+1})$	$O(bm)$
Optimality	Guaranteed	Not guaranteed
Completeness	Complete	Complete (with depth limit)
Memory Usage	High	Low

due to following a search down an individual branch until it backtracked. The length of the 21-step solutions depends wholly on the order states are explored. In this case, a solution was produced by DFS but it was not optimal in length, 40% longer to be precise. DFS clearly illustrates the trade-off in exploration of depth versus quality of solution.

B. Time and Space Complexity Comparison

Table III provides a detailed comparison of the algorithmic properties.

Where b is the branching factor (2–4), d is the optimal solution depth (15), and m is the maximum search depth.

C. Performance Characteristics

BFS Performance: With $b \approx 3$ and $d = 15$, BFS explores approximately $O(3^{16}) \approx 43$ million nodes in the worst case. For the Rabbit Leap Problem, BFS visited around 100 states before finding the optimal solution, demonstrating efficient exploration of the 140-state space. The queue-based implementation requires storing all frontier nodes, leading to high memory usage.

DFS Performance: DFS explores paths to depth m before backtracking, with worst-case time complexity $O(b^m)$. The stack-based implementation requires only $O(bm)$ space, storing the current path and visited states. While memory-efficient, DFS may explore deeply suboptimal paths, as evidenced by the 21-step solution.

D. Practical Implications

For the Rabbit Leap Problem, BFS is the preferred algorithm due to:

- **Small State Space:** Only 140 theoretical states make exhaustive search feasible.
- **Optimality Requirement:** The problem explicitly requires the fewest steps.
- **Guaranteed Completeness:** BFS always finds a solution if one exists.

DFS would be more suitable for problems where:

- Memory constraints are severe.
- Any solution is acceptable (not necessarily optimal).
- The state space is very large but solutions exist at moderate depths.

VI. CONCLUSION

This analysis demonstrates the central trade-offs between breadth-first search (BFS) and depth-first search (DFS) when talking about state space search problems. BFS provides an optimal solution and guarantees completeness in producing a solution, while potentially incurring a larger memory footprint and state size. DFS is generally faster while using a minimal amount of memory per state, but ultimately results in a lower quality of solutions. Applying this to the context of the Rabbit Leap Problem, it is easy to see why BFS is the clear algorithm to use here, when given the small size of the state space, and the demand for an optimal solution. The extra distinction of a viable 15-step long optimal solution versus the DFS solution of 21 steps provides a clear practical example of the importance of choosing the right algorithm based on the characteristics of the problem, and solution space limitations.

This implementation exercise re-emphasized the key concepts from the reference, especially about uninformed search strategies, and their performance characteristics in varied problem domains.

ACKNOWLEDGMENT

The authors would like to thank Dr. Pratik Shah, faculty of IIIT Vadodara for their guidance and support in conducting this analysis.

REFERENCES

- [1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Hoboken, NJ, USA: Pearson, 2020.