LAXMI CHARITABLE TRUST'S
SHETH L.U.J. COLLEGE OF ARTS & SIR M.V. COLLEGE OF SCIENCE & COMMERCE
DR. S. RADHAKRISHNAN MARG, ANDHERI(EAST), MUMBAI - 400069

Department of Computer Science

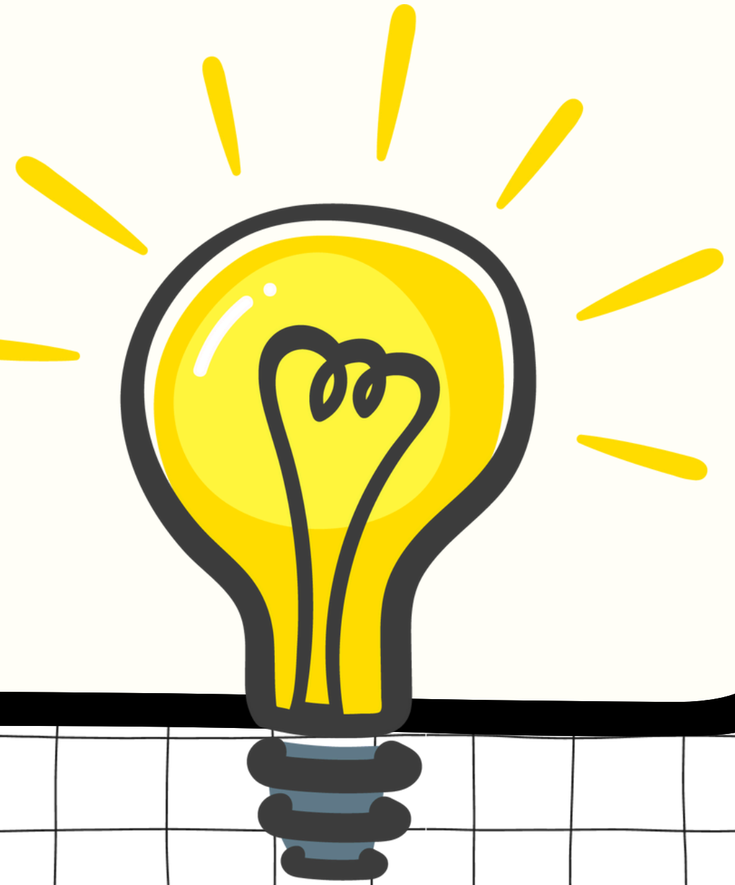
Snake And ladder Problem

Name : Shreeraj Desai, Kunal Joshi, Ghanshyam Kanojiya, Shobit Halse
Class : S. Y. B. Sc. Computer Science
Roll No. : S075,S086,S087,S083
Semester : Semester III
Academic year : 2024-2025

Link : https://colab.research.google.com/drive/1kM2GAXBHQxY4FniGnDfhByc1wzlf7nlS?usp=sharing

Subject Incharge : Dr. Mahendra Kanojia

Snake And Ladder



Contents

Introduction

Snake and Ladder is a game played on a numbered board, where players roll dice and move forward based on the number rolled. The board contains snakes and ladders, with the goal being to reach the final square, square 100, before anyone else.

Why is it considered a problem?

- Randomness
- Interdependence
- Conditional Probabilities
- Complexity

Solution Approaches

BFS(Breadth-First Search)

Explores all neighbors at the present depth before moving to nodes at the next depth level

DFS(Depth-First Search)

Explores as far as possible along each branch before backtracking.

Priority Queue

Finds the shortest path between nodes in a graph, prioritizing the exploration of the most promising nodes first

Priority Queue Solution for the Snake and Ladder Problem

In this solution, we can model the game as a weighted graph where each cell on the board is a node, and moving from one cell to another has a uniform "weight" (the number of dice rolls). Since we are interested in finding the minimum number of dice rolls to reach the last cell, the problem becomes finding the shortest path in a graph, which can be efficiently solved using a priority queue.

Algorithm Steps:

Initialization:

- Create a moves array to track ladders and snakes.
- Create a priority queue (min-heap) where each entry is a tuple (cost, current_node).
- Create a visited array to ensure each node is processed only once.

Priority Queue Processing:

- Pop the node with the least dice rolls from the priority queue.
- For each dice roll (from 1 to 6), compute the next position. If a snake or ladder is present, adjust the next position accordingly.
- Push the next position along with the incremented dice rolls (cost) into the priority queue.

Termination:

- The algorithm terminates when the destination (last node) is reached. At that point, the number of dice rolls corresponds to the shortest path (minimum dice rolls).

Priority Queue Solution for the Snake and Ladder Problem Code Implementation:

```python
# Example Board where -1 indicates no snake/ladder, and other values represent the destination
board = [-1, -1, -1, -1, 14, 20, -1, -1, -1, -1, -1, 4, -1, -1, 19, -1, -1, -1, -1, -1, -1]
```

```python
import heapq

def priority_queue_solution(board):
    n = len(board)  # Number of cells on the board

    # Initialize the moves array to handle snakes and ladders
    moves = [-1] * n
    for i in range(n):
        if board[i] != -1:
            moves[i] = board[i] - 1  # Convert 1-based to 0-based indexing

    # Priority queue where each entry is (number_of_moves, position)
    pq = [(0, 0)]  # Start from the first cell (position 0) with 0 moves
    visited = [False] * n
    visited[0] = True

    while pq:
        # Pop the position with the minimum number of moves (cost)
        cost, curr = heapq.heappop(pq)

        # If we reach the last cell, return the number of moves (cost)
        if curr == n - 1:
            return cost

        # Explore the next 6 possible positions (dice rolls from 1 to 6)
        for dice in range(1, 7):
            next_pos = curr + dice

            if next_pos < n:
                if moves[next_pos] != -1:  # If there's a ladder or snake
                    next_pos = moves[next_pos]

                if not visited[next_pos]:
                    visited[next_pos] = True
                    heapq.heappush(pq, (cost + 1, next_pos))

    return -1  # If the end is not reachable
```

Example Board Input:

```
# Example Board where -1 indicates no snake/ladder, and other values represent the destination
board = [-1, -1, -1, -1, 14, 20, -1, -1, -1, -1, -1, 4, -1, -1, 19, -1, -1, -1, -1, -1, -1]
```

Output:

```
Minimum number of dice rolls to reach the end: 2
```

Time and Space Complexity:

1. **Time Complexity:**
   - Each node (cell) can have at most 6 neighbors (due to the dice roll), and we push each neighbor into the priority queue once. Since there are n cells, this gives us a time complexity of O(n log n), where the log n factor comes from maintaining the priority queue.
2. **Space Complexity:**
   - The space complexity is O(n) because we maintain arrays (moves and visited) of size n, and the priority queue can store at most n elements.

Depth-First Search (DFS) Approach:

Overview:

- Depth-First Search (DFS) is an algorithm used to explore the game board of Snake and Ladder, treating it as a graph where each position is a node and possible moves (normal, via ladders, or down snakes) are edges. Here's how DFS is applied in the context of the game:

- DFS reaches a position with no unvisited neighbors, it backtracks to the previous position to explore any remaining unvisited paths.

- **DFS explores all possible moves from a starting position, simulating player progression.**
- **Adjusts position for snakes (down) and ladders (up) while exploring paths.**

- The goal of DFS in the Snake and Ladder game is to explore all possible paths from the starting position to the goal (usually position 100), considering normal moves, snakes, and ladders.
- DFS ensures all possible paths and moves are considered, providing insights into the game's structure and dynamics.
- DFS backtracks when encountering revisiting positions, ensuring it explores all possible routes, even after encountering snakes that send players back to earlier positions.

Code Example (DFS Implementation)

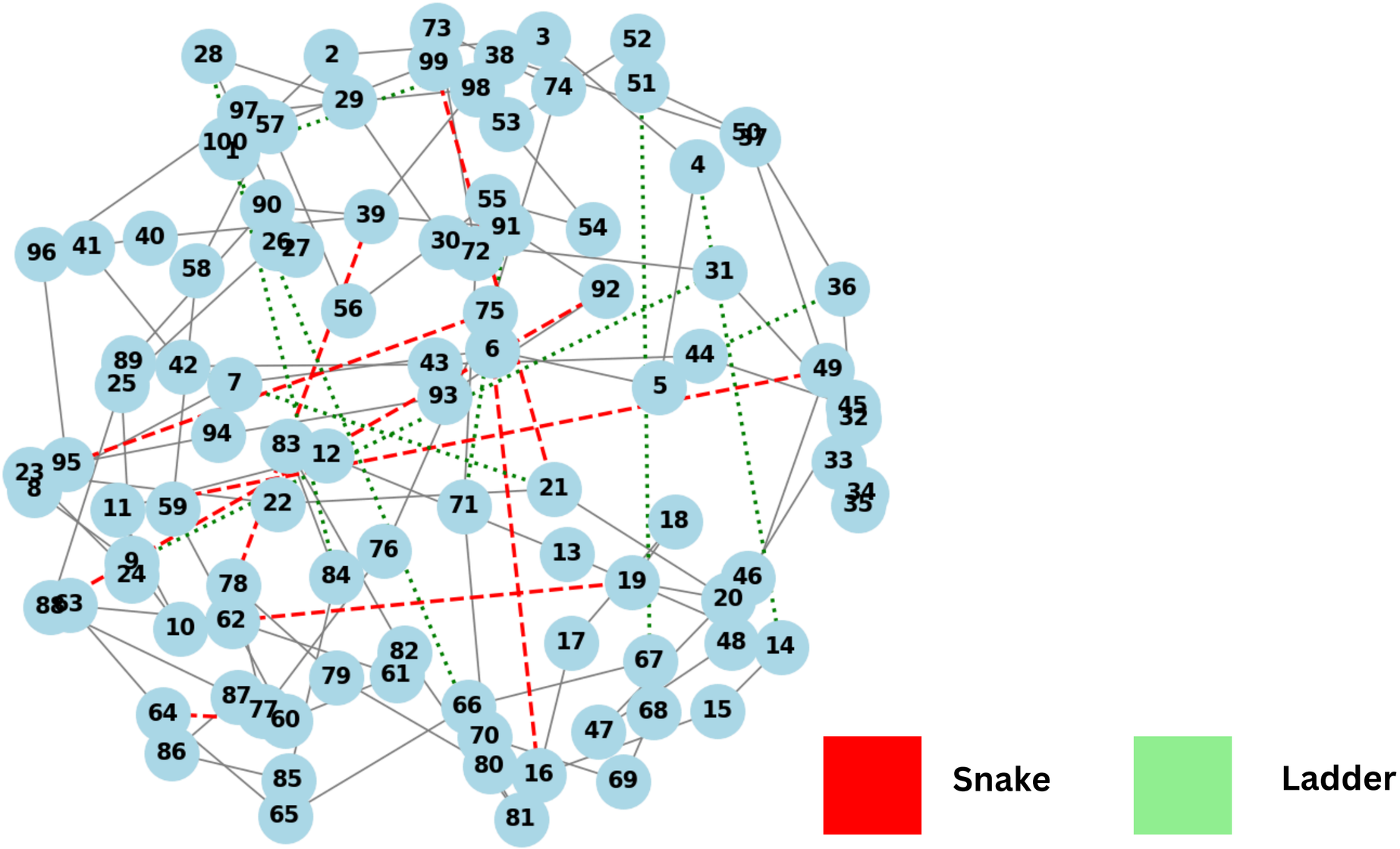
```python
def dfs_solution(board):
    n = len(board)
    (variable) visited: list[bool]
    visited = [False] * n

    for i in range(n):
        if board[i] != -1:
            moves[i] = board[i] - 1

    def dfs(pos):
        if pos == n - 1:
            return 0  # Reached the last cell

        visited[pos] = True
        min_moves = float('inf')

        for dice in range(1, 7):
            next_pos = pos + dice
            if next_pos < n and not visited[next_pos]:
                if moves[next_pos] != -1:
                    next_pos = moves[next_pos]
                min_moves = min(min_moves, 1 + dfs(next_pos))

        visited[pos] = False
        return min_moves

    result = dfs(0)
    return result if result != float('inf') else -1

board = [-1, -1, -1, -1, 14, -1, -1, -1, -1, -1, -1, 4, -1, -1, 19, -1, -1, -1, -1, -1, -1]

result = dfs_solution(board)
print(f"Minimum number of dice rolls to reach the end: {result}")
```

Core Code for the Minimum Number of Dice Rolls

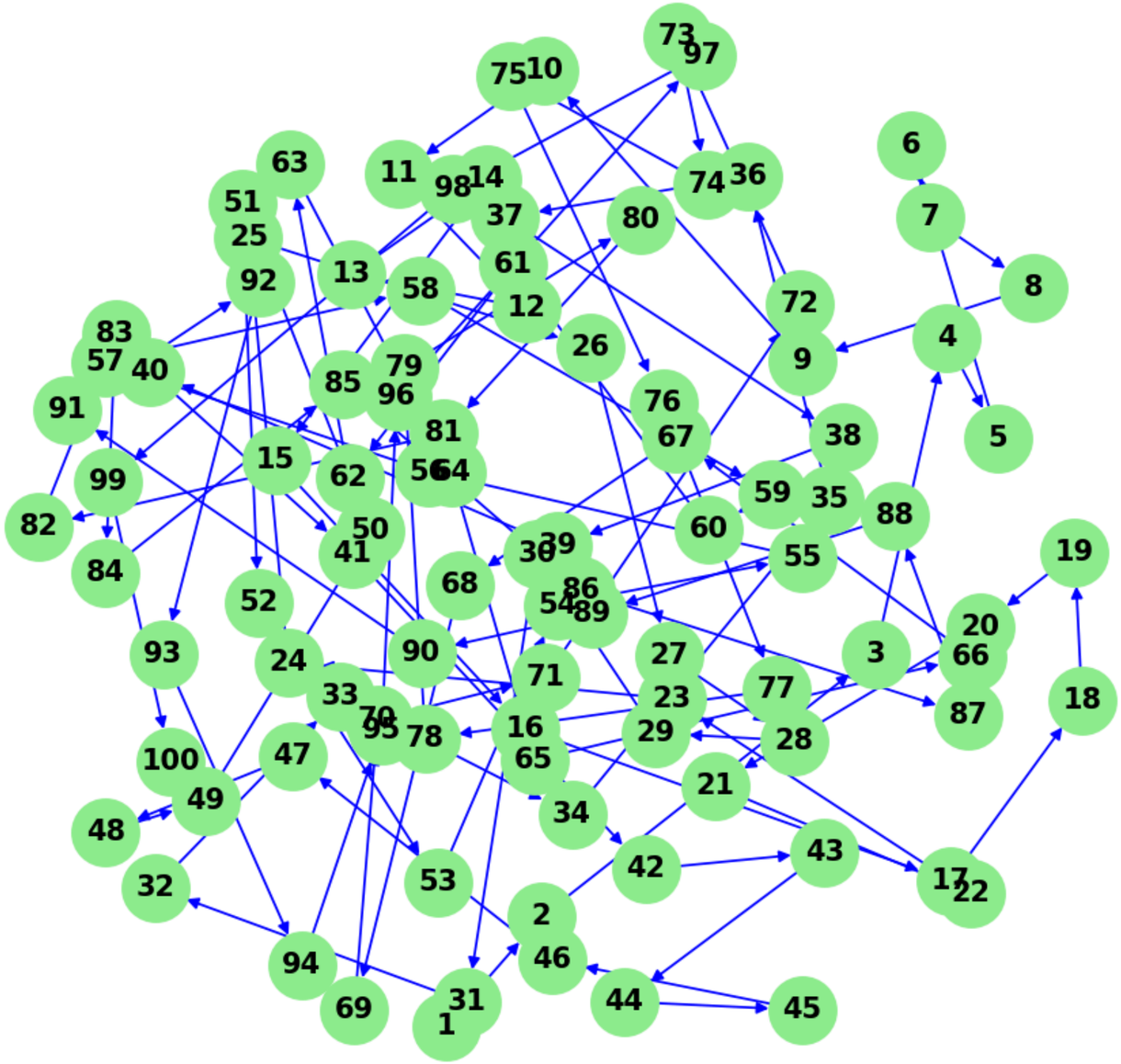
Graph Representation

Snake and ladder graph

DFS Graph



Snake and ladder graph


73
97
75
10
6
63
11
98
14
74
36
51
7
25
37
80
61
13
92
58
8
12
72
83
4
57
40
26
9
79
85
96
76
91
81
67
38
5
15
62
56
64
99
59
35
88
82
50
60
39
41
30
55
19
84
68
86
52
54
89
20
93
24
90
27
3
66
71
77
33
23
18
87
16
29
28
100
47
78
65
21
49
34
48
43
42
17
32
53
22
2
94
46
69
31
44
45
1

DFS Graph

Breadth-First Search (BFS) Approach:

Overview:

- BFS is a simple and effective algorithm for finding the shortest path in unweighted graphs, which fits the Snake and Ladder problem where each move (dice roll) is essentially an unweighted edge.

- It explores all nodes at the current distance level before moving on to the next, ensuring that the first time you reach the target, it’s through the minimum number of moves.

- In Snake and Ladder, the goal is to reach the final square in the minimum number of dice rolls.
- Each square is connected to up to 6 other squares (due to dice rolls), and BFS is perfect for finding the shortest path in this type of graph structure.
- The use of BFS guarantees the fewest dice rolls to reach the destination, as BFS explores all possible paths level by level (similar to moves in Snake and Ladder).

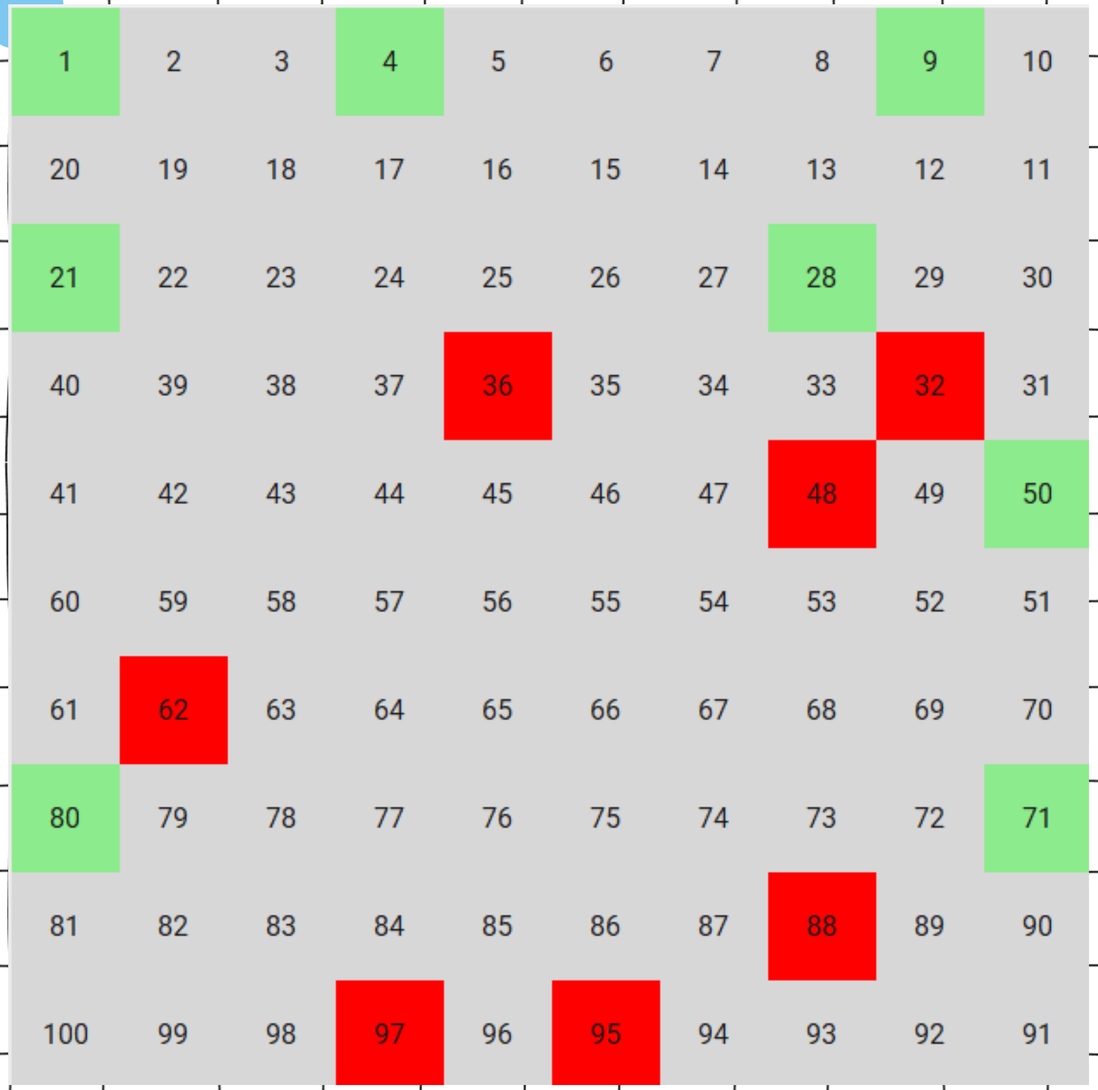
Code Example (BFS Implementation)

```python
from collections import deque
def bfs_solution(board):
    n = len(board)
    moves = [-1] * n  # Initialize moves array
    # Preprocess the board to handle snakes and ladders
    for i in range(n):
        if board[i] != -1:
            moves[i] = board[i] - 1
    # BFS setup
    queue = deque([0])  # Starting from position 0
    visited = [False] * n
    visited[0] = True
    distance = 0
    while queue:
        size = len(queue)
        for _ in range(size):
            curr = queue.popleft()

            if curr == n - 1:
                return distance  # Reached the last cell
            # Explore next 6 positions
            for dice in range(1, 7):
                next_pos = curr + dice
                if next_pos < n and not visited[next_pos]:
                    visited[next_pos] = True
                    if moves[next_pos] != -1:
                        queue.append(moves[next_pos])
                        print(queue)
                    else:
                        queue.append(next_pos)
        distance += 1

    return -1  # If not reachable
board = [-1, -1, -1, -1, 14, 20, -1, -1, -1, -1, -1, 4, -1, -1, 19, -1, -1, -1, -1, -1, -1]
result = bfs solution(board)
```

BFS Solution for the Snake and Ladder Problem Code Implementation:

Snake

Ladder



Ladder from 1 to 38
Ladder from 4 to 14
Ladder from 9 to 30
Ladder from 21 to 42
Ladder from 28 to 76
Ladder from 50 to 42
Ladder from 71 to 67
Ladder from 80 to 99

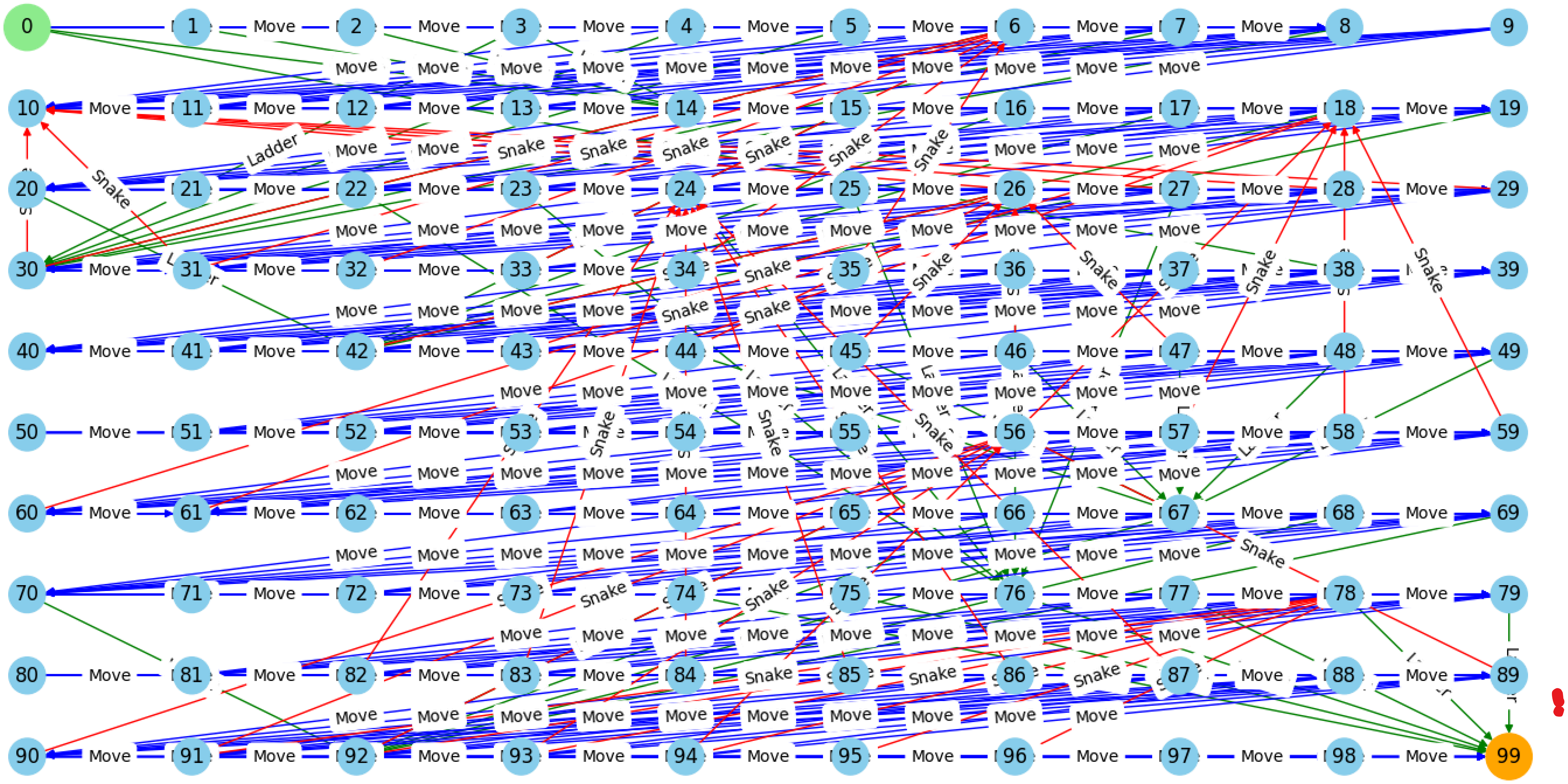
Snake from 36 to 6
Snake from 32 to 10
Snake from 48 to 26
Snake from 62 to 18
Snake from 88 to 24
Snake from 95 to 56
Snake from 97 to 78

The least number of dice rolls needed: 5

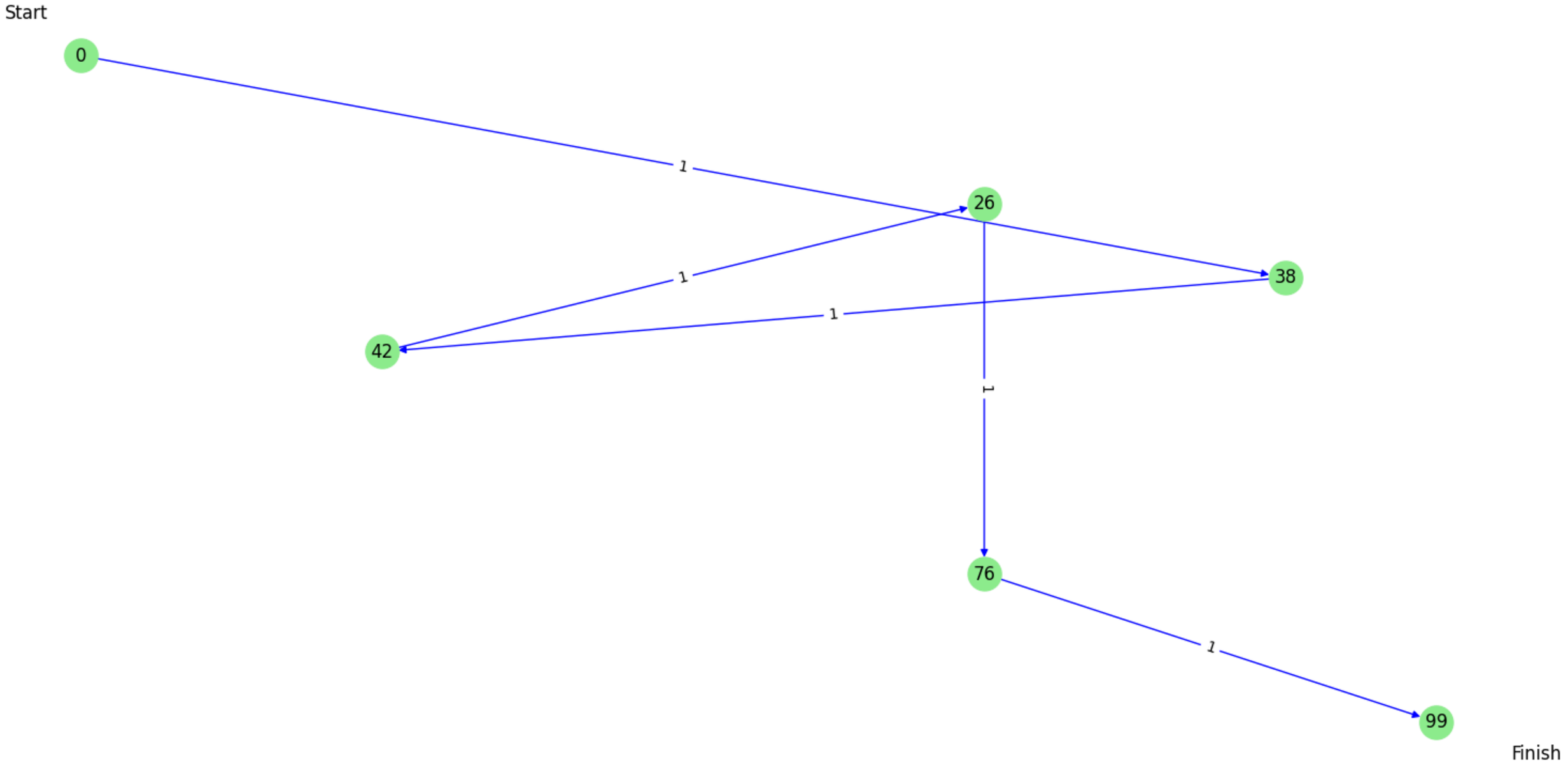
Graph Representation

All possible rolls

Optimal Graph



All possible rolls

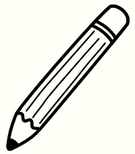


Optimal Graph

Conclusions

(N)=Number of Squares

Priority Queue (Dijkstra's approach)



Priority Queue works, but it's unnecessarily complex for this problem since all dice rolls have equal weight.

Time Complexity : O(*N* log *N*)

BFS

BFS is the most efficient approach, as it guarantees the shortest path with minimal complexity.

Time Complexity : O(*N*)

DFS

DFS is not suitable for finding the shortest path because it doesn't explore the most efficient route.

Time Complexity : O(6^*N*)

Thank you