Jumping Jim Maze Solver Report

Lijing Li

1 Introduction

In this report, I present a graph-based solution to the Jumping Jim problem. Jim has to find the shortest jumping path from the top-left to the bottom-right cell of given maze, where each cell contains a number indicating the allowed jump distance in E,W,S,N four directions. I model the problem using a directed unweighted graph, and solve it using the BFS algorithm, which guarantees the shortest path in such graph.

2 Graph Model

2.1 Graph Model Type

I chose the directed unweighted graph to present this maze. Because it can present the directions jumps between cells, and for this question we don't need to consider the weight of each path, just need to find the shortest jump path, not the minimum total cost.

Vertex :

Each cell at position (r,c) in the maze corresponds to a unique vertex in the graph. The index maintains the spatial relationships with the maze size of rows and cols:

index = r * col + c + 1

r = (index - 1) // n

c = (index - 1) % n

For example, in a 3×3 grid:

index 1 = (0,0), index 2 = (0,1), ..., index 9 = (2,2)

- Edge:

Each directed edge represents a valid jumps from cell (i,j) with value:

E: (i, j) → (i, j+v), if j+v < cols

W: (i, j) → (i, j-v), if j-v ≥ 0

S: (i, j) → (i+v, j), if i+v < rows

N: (i, j) → (i-v, j), if i-v ≥ 0

Each edge corresponds to one legal move, and only in-bounds jumps are considered.

2.2 Graph Construction (Pseudocode)

```
for r in 0 to rows - 1
for c in 0 to cols - 1
v = maze[r][c]
from = r * cols + c + 1

# Move East
if c + v < cols:
    add edge (i,j)→(i,j+v);
```

```
# Move West
if c - v >= 0:
    add edge (i,j)→(i+v,j);

# Move South
if r + v < rows:
    add edge (i,j)→(i+v,j);

# Move North
if r - v >= 0:
    add edge (i,j)→(i-v,j);
```

The graph is constructed by iterating over each cell in the maze. If a jump in any direction stays within the maze bounds, I create a directed edge from the current cell to the target cell.

2.3 3x3 Example

I give a 3x3 maze to help explain:

```
|1  1  2|
|4  2  1|
|3  1  0|
```

Jim starts at index 1 with a jump value of 1. From here, he can jump East 1 step to index 2, or South 1 step to index 4. There are two possible directed edges for index 1: (1→2) and (1→4).

The following list contains all the valid directed edges used to construct the graph from the given maze.

```
directed_edges = [
    (1, 2), (1, 4),                 # From index 1: E, S
    (2, 1), (2, 3), (2, 5),         # From index 2: W, E, S
    (3, 1), (3, 9),                 # From index 3: W, S
    //Index 4,5, the i,j±value > n, which Jim will jump off the maze, so they cannot be considered.
    (6, 3), (6, 5), (6, 9),         # From index 6: W, N, S
    (8, 5), (8, 7), (8, 9)          # From index 8: N, W, E
]
```

This example demonstrates how valid edges are generated only when the jump remains within bounds. The resulting graph allows BFS to be applied for finding the shortest path from index 1 to index 9.

3 Algorithm Design

3.1 BFS

I selected the BFS algorithm, because it handles directional paths without needing edge weights and always finds the shortest path in unweighted graphs.

```
function BFS(start, goal):
    queue ← new Queue of paths
    visited ← empty set

    enqueue [start] into queue
    add start to visited

    while queue is not empty:
    path ← dequeue from queue
    last ← last node in path

    if last == goal:
        return path

    for each neighbor in adjacency list of last:
    if neighbor not in visited:
        mark neighbor as visited
        newPath ← copy of path with neighbor appended
        enqueue newPath into queue

    return null
```

The BFS algorithm begins at the start vertex and explores all reachable neighbors level by level. A queue is used to manage the frontier of exploration, and a visited set ensures that each vertex is only explored once. Once the goal node is reached, the algorithm returns the path taken. Since the graph is unweighted, the first time the goal node is encountered corresponds to the shortest path.

3.2 Graph Direction

To represent the final path in terms of directions E, W, N, S, an additional function path() is needed to convert the index-based path into directional steps.

```
function getDirection(path, rows, cols):
for i from 1 to path.length - 1:
    prev = path[i - 1]
    curr = path[i]
    (r1, c1) = ((prev - 1) / cols, (prev - 1) % cols)
    (r2, c2) = ((curr - 1) / cols, (curr - 1) % cols)

    if r2 == r1 and c2 > c1:
        print "E"
    else if r2 == r1 and c2 < c1:
        print "W"
```

```
        else if c2 == c1 and r2 > r1:
            print "S"
        else if c2 == c1 and r2 < r1:
            print "N"
```

Applied to the earlier example:

|1 1 2|
|4 2 1|
|3 1 0|

The shortest path found using BFS is [1, 2, 3, 9]. We convert this to directions using the method described above:

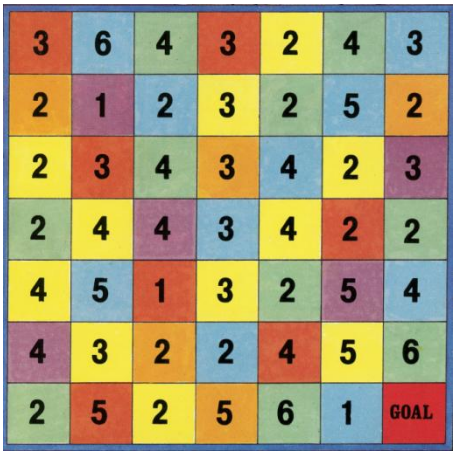
1 → 2: move East

2 → 3: move East

3 → 9: move South

Therefore, the final route is: E E S

4 Jumping Jim



Using the graph model and the BFS algorithm described earlier, we can construct the graph representation of the maze and compute the shortest jump path for Jim.

As shown in the Jim-output.txt, Jim would follow the path: E S S N S W E N W E E W S E N W S E.