Vamshi Jandhyala

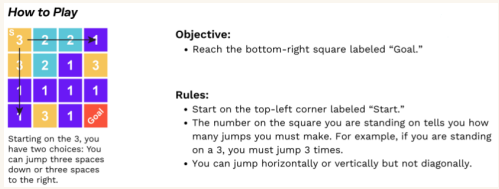
Solving the Jumping Julia Maze

✠

December 2024

The Jumping Julia Maze from the Julia Robinson Mathematics Festival is a puzzle in which the player navigates from a starting square to a goal by making jumps determined by the numbers in a grid. The natural framing is a shortest-path problem on a directed graph. We build the graph in Python, solve it with NetworkX, and visualise the solution with matplotlib.

Problem



Solution approach

1. Graph representation

The key insight is to model the puzzle as a graph:

- Each square in the grid becomes a node.
- Valid jumps become directed edges between nodes.
- Finding a solution becomes a shortest-path problem.

The advantages: a natural representation of connected positions, well-established algorithms for path finding, and straightforward visualisation.

2. Implementation

```
class JumpingMaze:
    def __init__(self, grid):
        self.grid = np.array(grid)
        self.rows, self.cols = self.grid.shape
        self.G = self._create_graph()
```

```
def _create_graph(self):
    G = nx.DiGraph()
    for i in range(self.rows):
        for j in range(self.cols):
            G.add_node((i, j), pos=(j, -i))

    for i in range(self.rows):
        for j in range(self.cols):
            jumps = self.grid[i, j]
            if jumps == 0:     # goal has no outgoing edges
                continue
            for dx in [-jumps, jumps]:
                new_j = j + dx
                if 0 <= new_j < self.cols:
                    G.add_edge((i, j), (i, new_j))
            for dy in [-jumps, jumps]:
                new_i = i + dy
                if 0 <= new_i < self.rows:
                    G.add_edge((i, j), (new_i, j))
    return G
```

Each node carries its (row, col) coordinates; edges are created only for exact jump distances.

3. Finding the solution

```
def find_solution(self, start=(0, 0)):
    goal = (self.rows - 1, self.cols - 1)
    try:
        return nx.shortest_path(self.G, start, goal)
    except nx.NetworkXNoPath:
        return None
```

NetworkX's `shortest_path` efficiently returns the fewest jumps to the goal, or signals that no path exists.

4. Visualisation

The visualisation uses light blue for regular squares and light green for path squares; gray curved arrows for all jumps and bold red curved arrows for the solution; grid values inside each square and step numbers along the path.

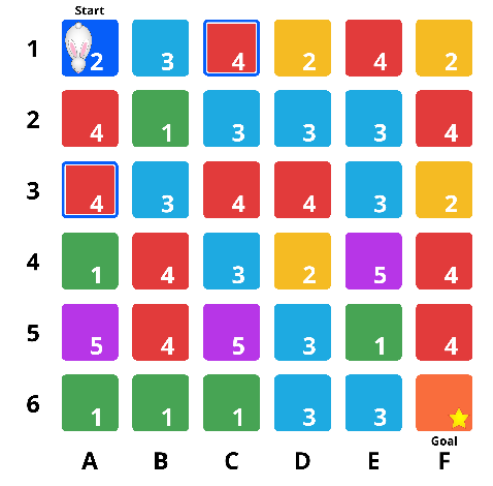
Usage

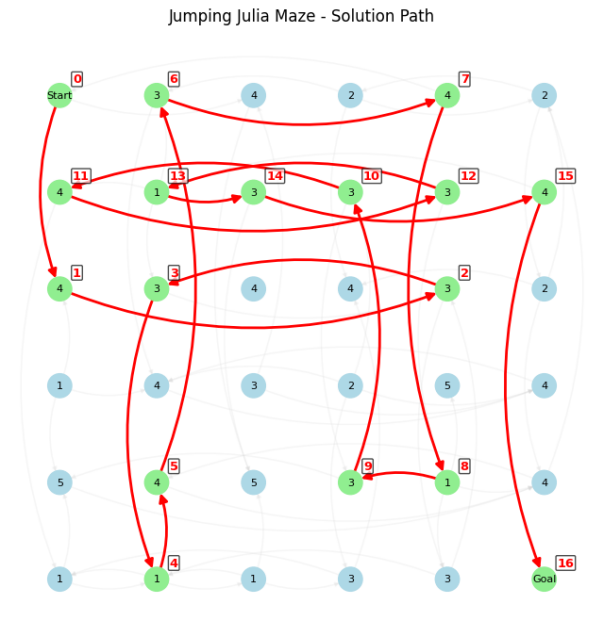
```
grid = [
    [3, 2, 2, 1],
    [3, 2, 1, 3],
    [1, 1, 1, 1],
    [1, 3, 1, 0],
]

maze = JumpingMaze(grid)
```

```
path = maze.find_solution()
if path:
    maze.visualize(path)
else:
    print("No solution exists!")
```

A harder maze





Conclusion

The Jumping Julia Maze illustrates how graph theory cleanly dispatches a path-finding puzzle. Converting the grid to a graph lets a generic shortest-path routine do the work,

and the visualisation makes the solution legible at a glance.