🧭 BFS Maze Solver Documentation

Author: Zeyad Mohamed Abdelwahab
Title: Find the Shortest Path in a Maze using BFS and Pyamaze

✍️ Personal Note:

This project was my **first step toward understanding how to implement algorithms practically**. I built it about a week ago to learn how **BFS works in real scenarios** — not just in theory — and how to write real code that finds a path through a maze.
It helped me a lot to build confidence before diving into more advanced algorithms like Minimax.

📄 Description:

This program demonstrates how to use the **Breadth-First Search (BFS)** algorithm to find the **shortest path** from the bottom-right to the top-left corner in a randomly generated maze.
We use the `pyamaze` library to generate the maze and visualize the path.

🧰 Requirements:

- `pyamaze` library (Install via `pip install pyamaze`)

🚀 How It Works:

1. We generate a maze using `pyamaze`.
2. We use **Breadth-First Search (BFS)** to explore the maze from the starting point to the goal.
3. We keep track of explored cells and build the shortest path.
4. We visualize the maze and the path using an agent that moves step by step.

📌 Key Concepts:

- **BFS (Breadth-First Search):**
  It is an algorithm that explores all possible paths level by level. It guarantees the shortest path in an unweighted graph like a maze.
- **Frontier:**
  The queue that contains the cells we are about to explore.
- **Explored:**
  A list of cells that we have already visited.
- **bfsPath:**
  A dictionary that keeps track of where each cell came from — used later to reconstruct the path.

🧠 Code Breakdown:

```
python
from pyamaze import *
```

→ We import the `pyamaze` library for generating and visualizing the maze.

```
python
def BFS(m):
```

→ Define a function to perform BFS on maze `m`.

```
python
    start = (m.rows, m.cols)
```

→ The starting point is the bottom-right corner of the maze.

```
python
    frontier = [start]
    explored = [start]
```

→ `frontier` is the list (queue) of cells to explore.
→ `explored` keeps track of already visited cells to avoid repetition.

```python
    bfsPath = {}
```

→ This dictionary helps us remember from where we reached each cell — to later reconstruct the path.

```python
    while len(frontier) > 0:
        currCell = frontier.pop(0)
```

→ As long as there are cells to explore, we remove the first cell from the queue.

```python
        if currCell == (1, 1):
            break
```

→ If we reached the goal (top-left corner), we stop searching.

```python
        for d in 'ESNW':
            if m.maze_map[currCell][d]:
```

→ Check all directions (East, South, North, West) and see if movement is possible.

```python
                if d == 'E':
                    childCell = (currCell[0], currCell[1] + 1)
                elif d == 'W':
                    childCell = (currCell[0], currCell[1] - 1)
                elif d == 'N':
                    childCell = (currCell[0] - 1, currCell[1])
                elif d == 'S':
                    childCell = (currCell[0] + 1, currCell[1])
```

→ Calculate the position of the neighboring cell based on the direction.

```python
                if childCell in explored:
                    continue
```

→ Skip cells we've already visited.

```python
                frontier.append(childCell)
                explored.append(childCell)
                bfsPath[childCell] = currCell
```

→ Add the new cell to `frontier` and `explored`, and remember from where we reached it.

```python
    fwdPath = {}
    cell = (1, 1)
```

→ Start building the final path from goal to start.

```python
    while cell != start:
        fwdPath[bfsPath[cell]] = cell
        cell = bfsPath[cell]
```

→ Use `bfsPath` to backtrack from goal to start and build `fwdPath`.

```python
    return fwdPath
```

→ Return the path to be used for visualization.

🎮 Main Program:

```python
if __name__ == '__main__':
    m = maze(5, 7)
    m.CreateMaze(loopPercent=40)
```

→ Create a 5×7 maze with some loops.

```python
    path = BFS(m)
```

→ Find the shortest path using our BFS function.

```python
    a = agent(m, footprints=True, filled=True)
    m.tracePath({a: path})
```

→ Add an agent (player) to the maze and let it trace the path.

```python
    l = textLabel(m, 'Length of Shortest Path', len(path)+1)
```

→ Display the length of the path on screen.

```python
    m.run()
```

→ Start the GUI to view the maze and animation.

☑ Output:

- Maze appears with the shortest path traced from the bottom-right to the top-left.
- A label shows the total path length.

❓ Why BFS?

- It guarantees the **shortest path** in an unweighted maze.
- Unlike DFS, it doesn't get stuck in deep paths.
- A* or Greedy require a **heuristic**, which we don't need here.