Artificial Intelligence for Big Data – Indivial report

Coursework 1

Mihael Zlatev

Representation of the Maze

The maze is represented via 2D Array with **“0”** or **“1”** as it’s elements. **“0”** means that the node is free and there isn’t any obstacle, whereas **“1”** means there is an obstacle. The maze class has the following fields:

- N (number of rows in the 2D array)
- M (number of columns in the 2D array)
- Grid (2D array representing the maze)
- Array of obstacles (tuples representing where in the maze to populate **“1” node**)

To construct a new Maze, you need to provide:

1. N – positive integer
2. M – positive integer
3. Obstacles (can be None)

```
class Maze:
    def __init__(self, n, m, obstacles=None):
        if n <= 0 or m <= 0:
            raise ValueError('Invalid maze dimensions. N and M must be greater than
0')
        self.n = n
        self.m = m
        self.grid = np.zeros((n, m))
        self.set_obstacles(obstacles)
```

Why Breath-First search

- BFS is optimal for finding the **shortest path** in unweighted graphs or grids, as it explores all nodes at the present depth level before moving to the next. This approach of the algorithm makes it highly effective if the end node is “close” to the start node.
- Unfortunatelly, in our scenario that might not be the case, then BFS won’t be optimal and DFS might be faster in some cases. However, since the graph represented via the 2D array is unweighted and the end node can be anywhere in the grid, there is no difference in the performance of BFS and DFS.
- Also, In unweighted graph DFS could result in longer path to the end node. For this reason, BFS is preferred here as it guarantees optimality in path length.
- It’s easy to implement using a Queue. Since BFS operates in layers, it only needs to store nodes at the current depth level, which makes it relatively memory-efficient in our case.

The method is called bfs:

```
def bfs(maze, start_node, end_node):
```

Parameters:

1. Maze – instance of the Maze class
2. start_node – (x,y) tuple
3. end_node – (x,y) tuple

Visualization of the searching algorithm

For the visualization I'm using **t**he **"matplotlib"** python library. The Maze is represented as 8x8 square which makes it perfect for visualizing small Mazes (E.g 10x10 Maze) While it's not optimal for visualizing large Mazes, I think it's perfectly fine for showcasing the work of any search algorithm. Also, larger plot can be used if needed.

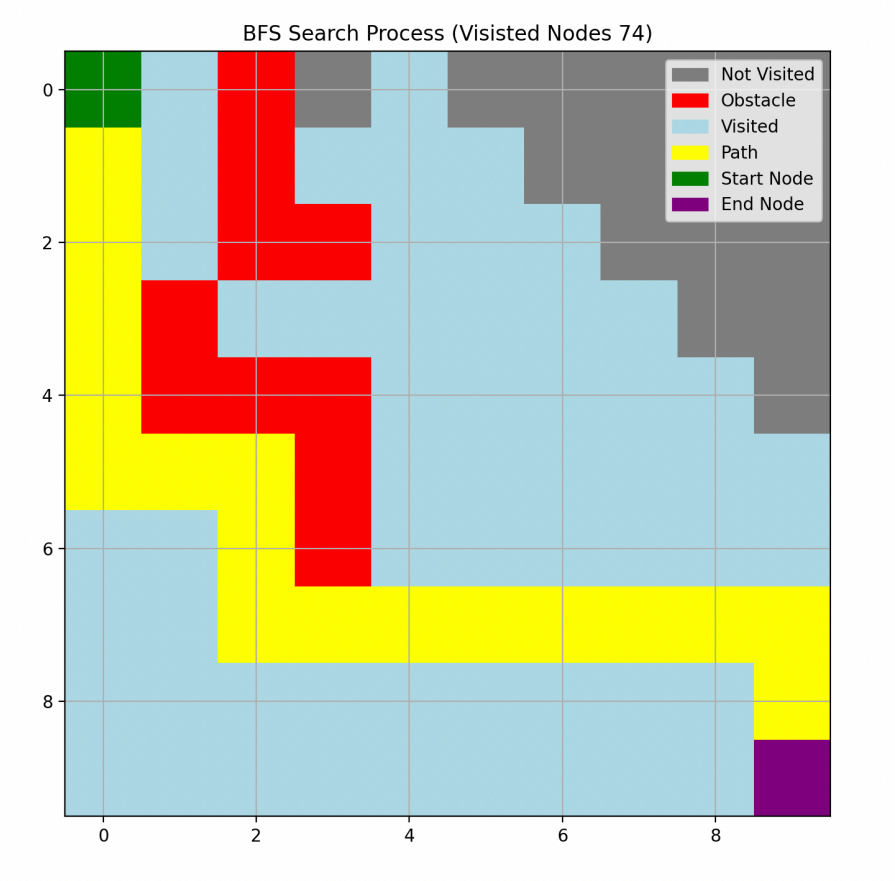
The method to visualize the work of BFS is called visualize_search:

```
def visualize_search(maze, start_node, end_node, path, visited_nodes)
```

Method Parameters:

1. maze – instance of the Maze class
2. start_node - (x,y) tuple
3. end_node – (x,y) tuple
4. path – array containing the path returned by BFS
5. visited_nodes – array containing the visited nodes by BFS

Example run with 10x10 Maze and obstacles at nodes:
[(0, 2), (1, 2), (2, 2), (2, 3), (3, 1),(4, 1), (4, 2), (4, 3), (5, 3), (6, 3)]



Time and space complexity

Breath-First search vs A-star algorithm

N – number of rows in the 2D Array (Maze grid)
M – number of columns in the 2D Array (Maze grid)

1. BFS (Breadth-First Search) on a 2D Grid

- In a grid-based search, BFS will explore each cell once (or possibly multiple times depending on the implementation and if there are revisits).
- The worst-case time complexity is ***O(N x M)*** because:
  - BFS will visit each cell once.
  - Since BFS uses a queue to explore all neighbors, it will expand to each node in layers until the goal is reached.
  - Space complexity is also ***O(N x M)*** BFS needs to store each node it visits (either in a queue or a visited set), so in the worst case, the queue could contain all nodes.

2. ***A-star Search on a 2D Grid****

- A* also has a worst-case time complexity of ***O(N x M)*** for a grid of size *NxM* as it might need to examine each cell once.
- However, A* uses a heuristic to guide its search, so in practice, it often visits fewer cells than BFS if the heuristic is good such as using Manhattan.

    - With a well-chosen heuristic, A* can often avoid exploring parts of the grid that do not lead toward the goal, thus reducing the actual number of nodes visited.
- Space complexity for A* is also ***O(N x M)***
    - A* uses a priority queue (open set) and a closed set to keep track of visited nodes and their costs.
    - In the worst case, these data structures can hold up to *NxM* nodes, meaning A* will need ***O(N x M)*** space.

Key Differences in Practice

- **BFS** explores blindly, meaning it will visit nodes uniformly outward from the start, often leading to unnecessary exploration.
- **A***, with a good heuristic, is more efficient in practice because it prioritizes nodes that seem closer to the goal, reducing unnecessary exploration and generally achieving faster results.