Warsaw University of Technology

FACULTY OF
ELECTRONICS AND INFORMATION TECHNOLOGY

EARIN

Laboratory 1, Group 1
Searching and optimization

Tymon Żarski 310992
Bartosz Peczyński 310703

WARSAW March 21, 2024

Contents

1. Introduction

Assigned task: Completion of the following laboratory involves developing a program capable of solving a maze, which is represented as a 2D grid featuring empty spaces, walls, a start position, and an end position.

The program is required to incorporate two distinct search algorithms: breadth-first search (BFS) and depth-first search (DFS), to find the path from the start to the end position. The primary objective is to calculate and return the number of steps necessary to reach the end from the start position.

Furthermore, the task required a mechanism for visualizing the search process, enabling step-by-step observation of the algorithm's progress through the maze. This visualization was created with the **pygame** library for efficient presenting of the process. The solution must be well documented, functioning correctly across a variety of test cases, including both standard scenarios and edge cases.

Algorithms used whire performing the task:

- **Breadth-First Search (BFS):** BFS is a fundamental search algorithm used primarily for traversing or searching tree or graph data structures. Starting from the root (or any arbitrary node in graphs), it explores the neighbour nodes first before moving to the next level neighbours. BFS is widely used for finding the shortest path on unweighted graphs, as it guarantees the minimum number of steps taken to reach a target node from the start. The main advantage of BFS is its ability to find the shortest path in terms of the number of edges traversed. However, its primary disadvantage is its memory consumption, as on top of visited nodes, it needs to store all the nodes of the current level before moving to the next level.
- **Depth-First Search (DFS):** DFS is next fundamental algorithm for traversing or searching a tree or graph. It differs from BFS by diving as deep as possible into the graph before backtracking to explore other branches. DFS is particularly useful for scenarios that require exploring all possible paths or configurations, such as puzzle solving, as it can be more memory efficient than BFS. The primary advantage of DFS is its lower memory requirements than BFS, as instead of storing all first-level child nodes, it stores the path to the currently investigated node, what depends more on depth than on breadth of the graph. However, its major disadvantage lies in its inability to guarantee the shortest path, as it may take detours deep into the graph before finding the target node.

2. Implementation

2.1. Architecture of the solution

Our solution is divided into four files: main.py, maze.py, visualize.py, tests.py. I this section we will go thought all of the files focusing on the **maze.py** which is the heart of the solution by implementing the BFS and DFS search. File **visualize.py** will explained based on the visual examples in the separate section **subsection 2.5**.

2.2. Step-by-step solution analysis

File main.py: This Python script is an entrypoint to our command-line application for generating a maze and visualizing the Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms. The script uses the **argparse** library to parse command-line arguments. It accepts three arguments:

- **–size** - used to specify the size of the maze (ex. 4 5 as 4 columns weight and 5 columns width),
- **–maze-file** - used to provide a pre-existing maze from a file (in such format that two first lines include start and finish point respectively and next lines contain 0 or 1 space separated indicating the maze details.
- **–screenshot-only** - used to force program to make only screenshot instead of showing the visualization.

To avoid any unwanted inputs or confusion the script checks if both **–size** and **–maze-file** are provided the program will display warning notifying that maze from **–maze-file** will be used.

When encountering the happy path if **–maze-file** is provided, it reads the start, finish positions and the maze itself from the file. If **–size** is provided, it generates a new maze of the specified size.

The script then prints the maze and the start and finish positions before the seraches begin (**Figure 2.3**), next it creates two Search objects, one for BFS and one for DFS, and performs the searches on the maze. It calculates the number of steps made by the algorithm and length of the final path for both BFS and DFS. Then it prints these values for the user to compare.

Lastly, program visualizes the data using the **visualize_data** function, which displays the maze and the paths found by BFS and DFS.

File maze.py: code within this file is a comprehensive implementation of both breadth-first search (BFS) and depth-first search (DFS) algorithms. The maze is represented as a 2D grid where each cell can either be an empty space (0) or a wall (1), later visited node (2). (Intentionally the values of the cells are kept in tn the emus to ensure code readability.) The goal is to find a path from a specified start position to a finish position. Here's a step-by-step breakdown focusing on the crucial parts, especially the search implementation and the BFS/DFS distinction.

2.3. Frontier Classes for DFS and BFS

The choice of frontier data structure is what fundamentally differentiates DFS and BFS.

BFS (Breadth-First Search) uses a queue data structure for the frontier, exploring all neighbors of a node before moving on to the neighbors' neighbors. This ensures that the search progresses uniformly across the breadth of the maze. The QueueFrontier class achieves this by removing nodes from the beginning of the list (First In, First Out - FIFO), ensuring that earlier discovered nodes are expanded first:

```
class QueueFrontier(StackFrontier):
    def remove(self):
        node = self.frontier[0]
        self.frontier = self.frontier[1:]
        return node
```

BFS is guaranteed to find the shortest path in unweighted graphs or mazes, which makes it particularly useful in many applications, albeit potentially more memory-intensive due to the broader exploration as mentioned in the previous **paragraph 1**

DFS (Depth-First Search) uses a stack data structure to manage the frontier, which stores the nodes that are pending exploration. In DFS, the algorithm goes as deep as possible into one branch before backtracking to explore other branches. This is achieved by always expanding the most recently discovered node (Last In, First Out - LIFO). The StackFrontier class represents this strategy by adding new nodes to the end of a list and removing from the end as well:

```
class StackFrontier:
    def add(self, node: Node) -> None:
        self.frontier.append(node)

    def remove(self) -> Node:
        node = self.frontier[-1]
        self.frontier = self.frontier[:-1]
        return node
```

This approach can be faster in finding a solution if the depth of the target node is relatively small compared to the breadth. However, it might not find the shortest path as explained in **paragraph 1**

2.4. Search Class

The Search class is where the maze is navigated using either the BFS or DFS algorithm, as determined by the **search_type** parameter. It is responsible for the following key operations:

```
class Search:
    def search(self):
        frontier = QueueFrontier() if self.search_type == "bfs" else StackFrontier()
        frontier.add(start_node)
        # Rest of the code
```

1. Initializing the search: Sets up the maze, start and finish points, and selects the search strategy (BFS or DFS),

2. Finding neighbors: Identifies possible next steps from the current location that are not walls and have not been visited, which is crucial for expanding the frontier. It is important to know that neighbors are always collected and added to the frontier in the same pattern/order to unsure the search integity.

```python

def find_neighbors(self, cell: tuple[int, int]) -> list[tuple[int, int]]:
    neighbors = []
    for i in range(1, -2, -1):
        for j in range(1, -2, -1):
            # Check and append logic omitted for brevity
    return neighbors
```

3. Marking cells as visited: Helps avoid revisiting the same cell, which is essential for efficiency and preventing infinite loops,
4. Performing the search: The core method where the algorithm iteratively expands nodes from the frontier, marks them as visited, and checks if the goal has been reached. This method demonstrates the essence of BFS and DFS:
   - In BFS, nodes are expanded in the order they were discovered, ensuring the shortest path is found in a maze without weighted paths,
   - In DFS, the search dives deep into one path until it hits a dead end or finds the goal, which can be more efficient in certain mazes but doesn't guarantee the shortest path.
5. Reconstructing the path: Once the goal is reached, the algorithm traces back from the goal to the start by following parent references, providing the sequence of steps taken to reach the goal.

The Search class embodies the application of BFS and DFS algorithms to solve practical problems, illustrating their differences, advantages, and limitations. While DFS might be more memory-efficient and faster in certain scenarios, BFS ensures the shortest path and is generally more reliable for pathfinding in unweighted graphs or mazes.

File tests.py This file is a set of unit tests for a maze search algorithm, leveraging Python's unittest framework to validate the Search class and the generate_maze function from a maze module. It assesses various scenarios, such as the reachability of positions, handling of invalid inputs, and the integrity of maze generation. The tests scrutinize the algorithm's response to different maze configurations, verifying correct pathfinding, appropriate error handling for edge cases, and the generation of mazes with specific characteristics. Testing from expected path validations to exception checks, this suite ensures the algorithm's consistency and error resilience, confirming its effectiveness across diverse situations. Search comparisoin tests will be further illustrated in the section **section 3**.

2.5. Results and search visualizations

Section describes a Python script using the Pygame library to visualize the process and results of comparing two pathfinding algorithms on a maze. The **visualize_data** function takes several arguments, including two sets of historical data representing the steps taken by the first and second algorithms, the grid size of the maze, the maze itself (as a 2D list), start and finish positions, and the final paths found by both algorithms.

Key features of the visualization include:

- Initializing Pygame and setting up a window with a specified size.
- Drawing the maze grid, walls, start, and finish positions using images loaded from an assets/images directory.

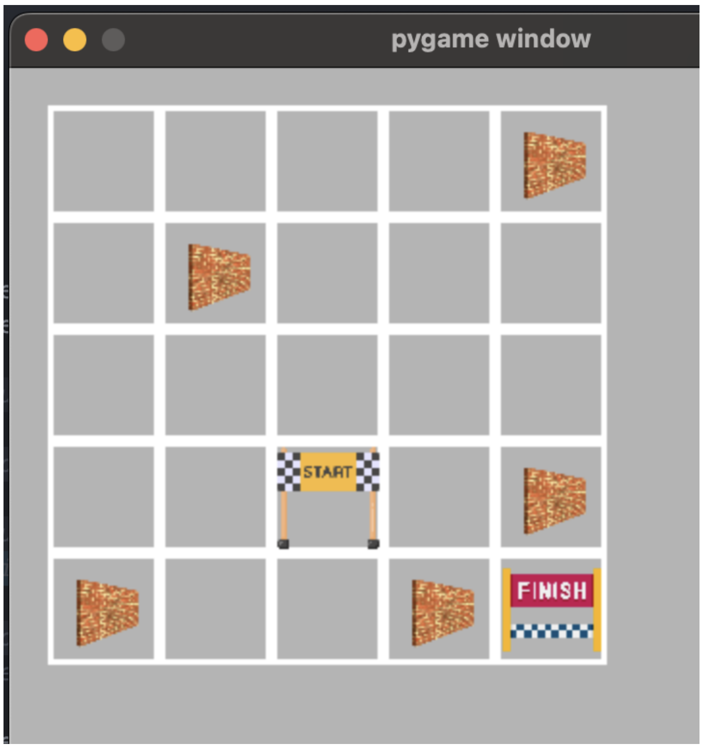


Figure 2.1. Initial draw of the maze before search begins

- Sequentially drawing the exploration steps of both algorithms on the maze grid, with different colors indicating the paths taken by the first and second algorithms. This is done by overlaying transparent surfaces on the Pygame window for each step, making the exploration process visually distinct.
  - Blue cell - denotes BFS search progress,
  - Red cell - denotes DFS search progress,
  - Purple cell - cells that has been explored both by BFS and DFS.
- Finally, drawing the paths that represent the solutions found by each algorithm, pausing between each step to highlight the progression from start to finish.
  - Black dot - denotes BFS final path,
  - White dot - denotes DFS final path,
  - Gray dot - cells that final path has in common between BFS and DFS.

The script is designed to run indefinitely until the user quits by closing the window or pressing the **q** key. This visualization tool provides an interactive and comparative way to analyze the efficiency, path length, and search patterns of two different pathfinding algorithms applied to the same maze.

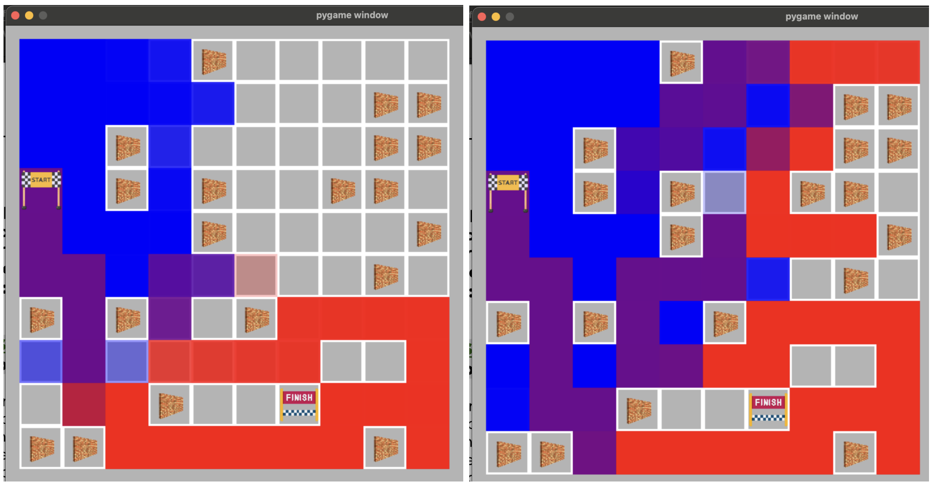
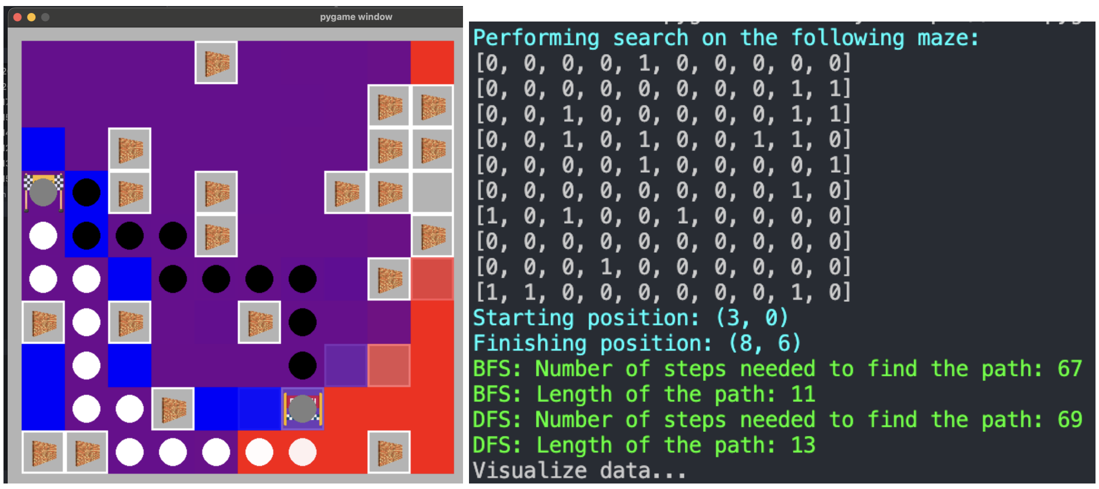


Figure 2.2. Enter Caption



```
Performing search on the following maze:
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 1]
[0, 0, 1, 0, 0, 0, 0, 0, 1, 1]
[0, 0, 1, 0, 1, 0, 0, 1, 1, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
[1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
[1, 1, 0, 0, 0, 0, 0, 0, 1, 0]
Starting position: (3, 0)
Finishing position: (8, 6)
BFS: Number of steps needed to find the path: 67
BFS: Length of the path: 11
DFS: Number of steps needed to find the path: 69
DFS: Length of the path: 13
Visualize data...
```

Figure 2.3. Presentation of found solutions

3. Discussion

Depending on the task instructions, discuss the results obtained by the solution, and explain them

Showcase different test cases if it is in the task, and explain the choice of these test cases

Both algorithms are able to reach the destination, when it is possible to do so. The main difference is the length of the found path and the number of steps made by the algorithm in order to find this path. DFS may not find the optimal path, as can be seen on Figure 3.1. The optimal path found by BFS is in this case of length equal to 8. DFS found longer path, which consists of 16 traversals between nodes. The traversed nodes also differ between those two algorithms. This can be especially seen on Figure 3.2. DFS found the optimal path in 7 steps, when BFS required 22 steps to complete this task. Those differences result from different strategy on selecting next node, which will be traversed. BFS first traverses nodes in the immediate vicinity. Only after checking all nodes, which are closest to the start node, it moves to next level. This allows

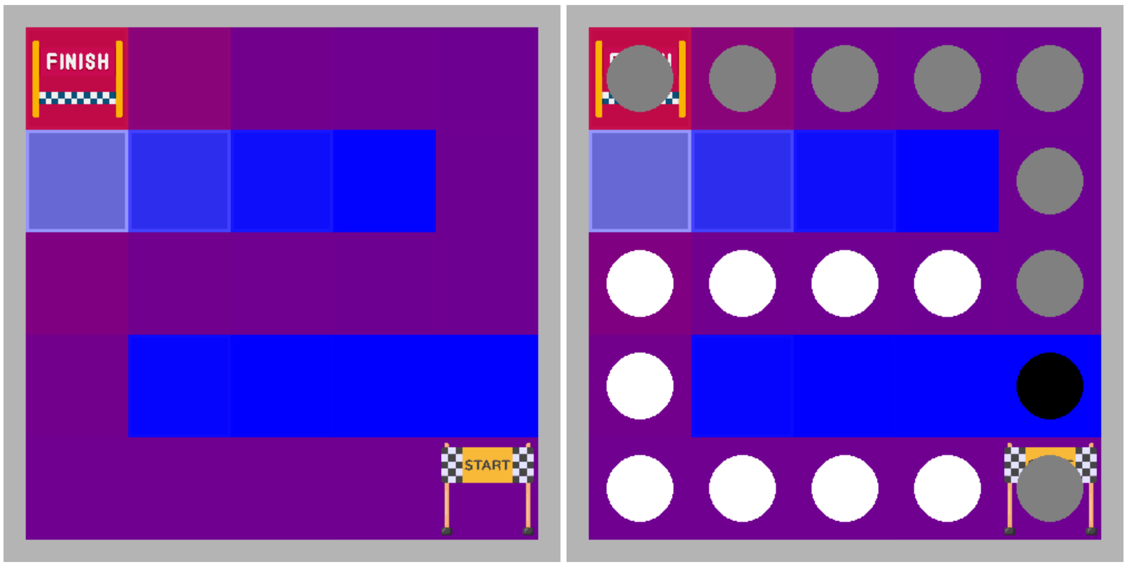


Figure 3.1. DFS gives sometimes longer paths than BFS

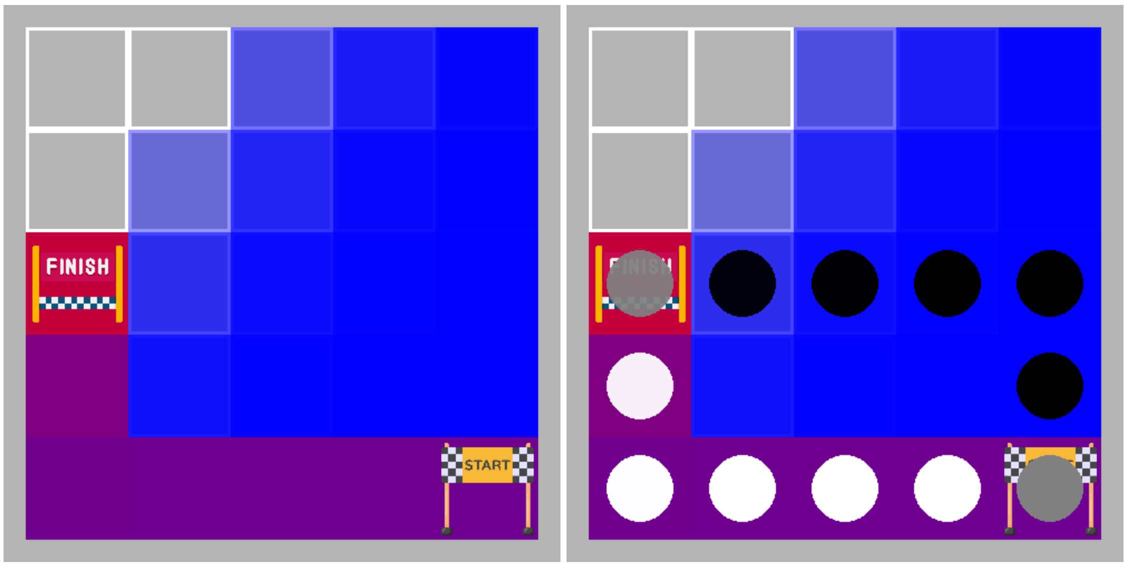


Figure 3.2. DFS sometimes makes less steps

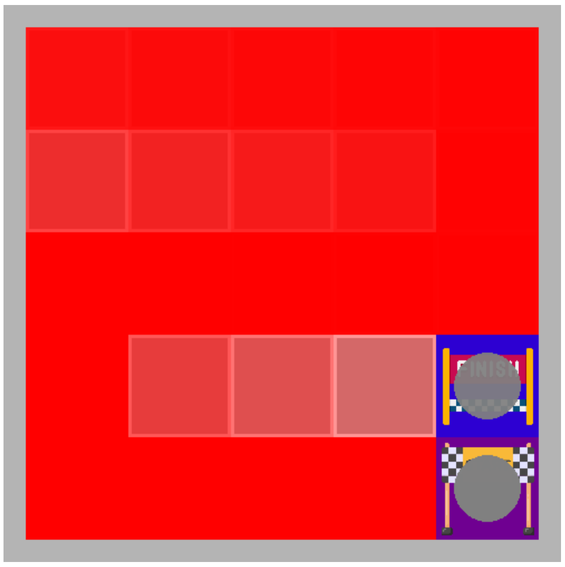


Figure 3.3. There are also situations, when dfs makes more steps than bfs

BFS to find shortest paths. In some cases though, BFS algorithm may traverse more nodes before finding the final node, because of the fact, that DFS may sometimes be ‘lucky‘ to find the path faster. It is also possible, that BFS will be faster in finding the path, which can be seen on Figure 3.3. The difference in steps made by the algorithms disappears, when it would be needed to traverse the whole graph. In this case, both algorithms would visit all nodes, but BFS is certain to give the shortest path. In most cases though, this problem does not need traversing all nodes.

4. Conclusion

Write conclusion including what have you learned, what was difficult and what can be done better

This laboratory made us see, that algorithms can have many applications, which are not directly connected to the tasks they were designed to solve. We used graph algorithms for solving a problem based on a square board, so the first step was to represent it as a graph. We refreshed our knowledge of BFS and DFS and applied it to our solution using pygame. This library was not used by us before, so we had to study its basics. There was no direct difficulty in solving the assigned problem, but the only issue is the amount of time we spent on writing the application, and parts not directly connected with algorithms. On the other hand, making it was quite interesting and we are thrilled to be able to make next tasks, which will contain more interesting algorithms.