Ministerul Educației și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Bonus Tasks

Visualization of Graph Traversal Algorithms

Elaborated:
st. gr. FAF-233 Moraru Patricia

Verified:
asist. univ. Fiștic Cristofor

Chișinău - 2025

Contents

Chapter 1

Introduction

Maze generation and solving algorithms are foundational in computer science for understanding graph traversal and optimization. This report focuses on the implementation and visualization of a maze generation algorithm (Kruskal's) and four maze-solving strategies (DFS, BFS, Dijkstra, and A*) using the Pygame library. Each technique provides insight into different aspects of pathfinding—from exhaustive search to heuristic-guided traversal.

Mazes are often represented as graphs, where each cell is a node and passages between them are edges. Generating a maze involves ensuring that all nodes are reachable (i.e., the graph is connected) while avoiding cycles to preserve the challenge of finding the correct path. Solving a maze, on the other hand, involves identifying a route from a defined start point to a goal node using one or more traversal techniques.

In this project, the maze is generated as a perfect maze—a type of maze with no loops and only one unique path between any two points. The generation phase is followed by the execution of different pathfinding algorithms that visualize their exploration strategies and ultimately compute a solution path.

This hands-on approach helps demonstrate how theoretical algorithms behave in practice. Moreover, the visual component enhances understanding of each algorithm's logic, strengths, and limitations when navigating a constrained environment. The educational value of such visual tools lies in their ability to illustrate algorithmic thinking in a concrete and engaging way.

Chapter 2

Methodology

The system is composed of two major components:

- **Maze Generation** using **Kruskal's Algorithm**, a classic algorithm for building minimum spanning trees, repurposed to generate perfect mazes.
- **Maze Solving** using four pathfinding algorithms that explore different search paradigms:
  - **Depth-First Search (DFS)** – a backtracking algorithm that goes as deep as possible before backtracking.
  - **Breadth-First Search (BFS)** – an exhaustive layer-by-layer exploration that guarantees shortest paths.
  - **Dijkstra's Algorithm** – a weighted shortest-path algorithm using uniform costs and priority queues.
  - **A* Algorithm** – an extension of Dijkstra's that adds heuristic guidance for faster convergence.

All components operate on a shared internal representation: a 2D grid-based maze where each cell corresponds to a node in a graph and connections between adjacent cells represent edges. The state of each cell is encoded using bit flags to track walls and visitation state. Rendering and animation are managed using the `pygame` library, which provides interactive visual feedback on each step of generation and traversal.

Maze Generation with Kruskal's Algorithm

Kruskal's algorithm is applied to a grid graph to create a perfect maze, meaning one without loops and with a unique path between any two points. The algorithm treats each cell as a node and all possible walls as weighted edges (although weights are not explicitly used in this implementation).

The process involves the following steps:

1. All cells are initialized in a separate disjoint set (using a Union-Find structure).
2. All walls between adjacent cells are listed and then shuffled randomly to introduce randomness into maze structure.

3. The algorithm iteratively examines each wall. If the wall separates two cells that belong to different sets (i.e., there is no path between them yet), the wall is removed and the sets are merged.

4. This merging continues until all cells belong to a single connected component and no cycles exist, forming a spanning tree.

5. During generation, the display is updated in real-time to show progress.

The final result is serialized and saved using Python's `pickle` module for use by the solvers.

Maze Solvers

After generation, the maze is solved by each of the four algorithms. All solvers operate under the assumption of equal edge costs and work on the same maze file, ensuring fairness in comparison.

- **Depth-First Search (DFS)** uses a stack-based approach. It aggressively explores one direction until it reaches a dead end, then backtracks to try alternative paths. It may not find the shortest path but is memory efficient.

- **Breadth-First Search (BFS)** uses a queue to visit all neighbors at a given depth before proceeding to the next level. It guarantees the shortest path in an unweighted graph and visually spreads out uniformly from the starting point.

- **Dijkstra's Algorithm** extends BFS by associating a distance cost with each node and always expanding the least-cost node next. Although unnecessary in an unweighted maze, it is included for comparative analysis with A*.

- **A* Algorithm** enhances Dijkstra's approach by integrating a heuristic (Manhattan distance) to estimate the cost from a node to the goal. This prioritizes nodes that are closer to the target, making it typically the most efficient in terms of speed and path optimality.

Each solver also visually highlights the explored path and the final route to the goal using colored lines and node animations. This provides intuitive insight into how each algorithm navigates the maze environment.

Chapter 3

Experiments and Results

3.1 Maze Generation: Kruskal's Algorithm

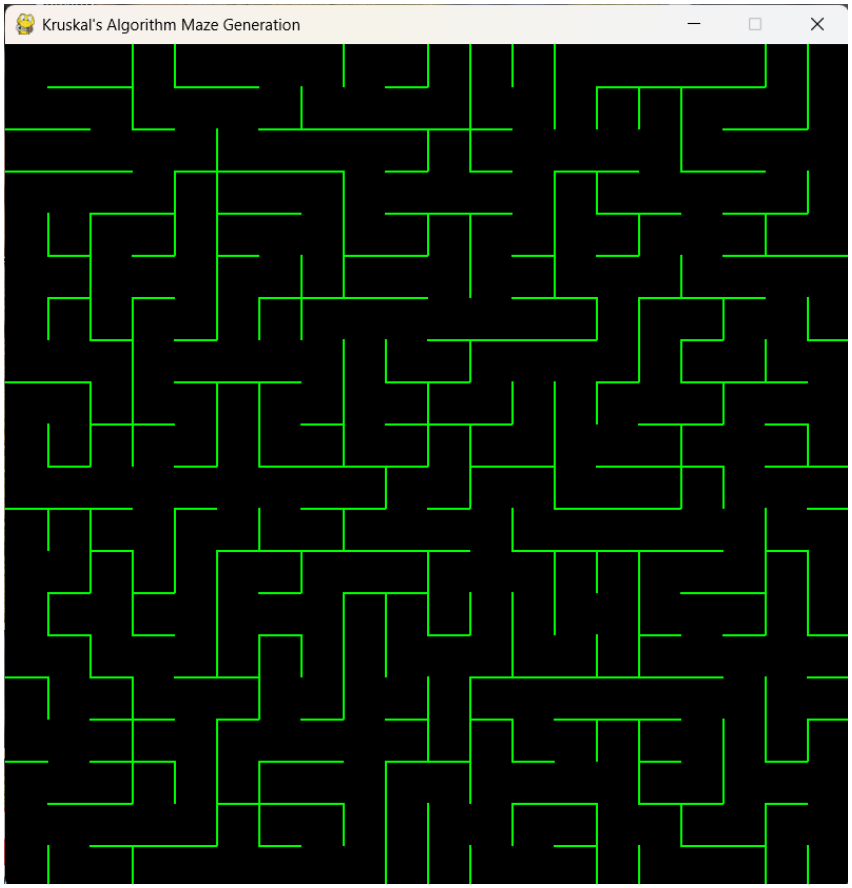

Figure 3.1: Maze generation using Kruskal's Algorithm

Listing 3.1: Maze generation - Kruskal's Algorithm

```
def generate_maze(self):
    dset = DisjointSet(self.rows * self.cols)
    walls = [(x, y, S) for y in range(self.rows) for x in range(self.
        cols) if y < self.rows - 1] + \
            [(x, y, E) for y in range(self.rows) for x in range(self.
                cols) if x < self.cols - 1]
```

```
    random.shuffle(walls)
    remaining_walls = set(walls)

    while walls:
        x, y, direction = walls.pop()
        nx, ny = x + DX[direction], y + DY[direction]
        if dset.find(y * self.cols + x) != dset.find(ny * self.cols +
            nx):
             dset.union(y * self.cols + x, ny * self.cols + nx)
             self.grid[y][x] |= direction
             self.grid[ny][nx] |= OPPOSITE[direction]
             self.grid[y][x] |= IN
             self.grid[ny][nx] |= IN
             remaining_walls.remove((x, y, direction))
```

3.2 Depth-First Search (DFS)

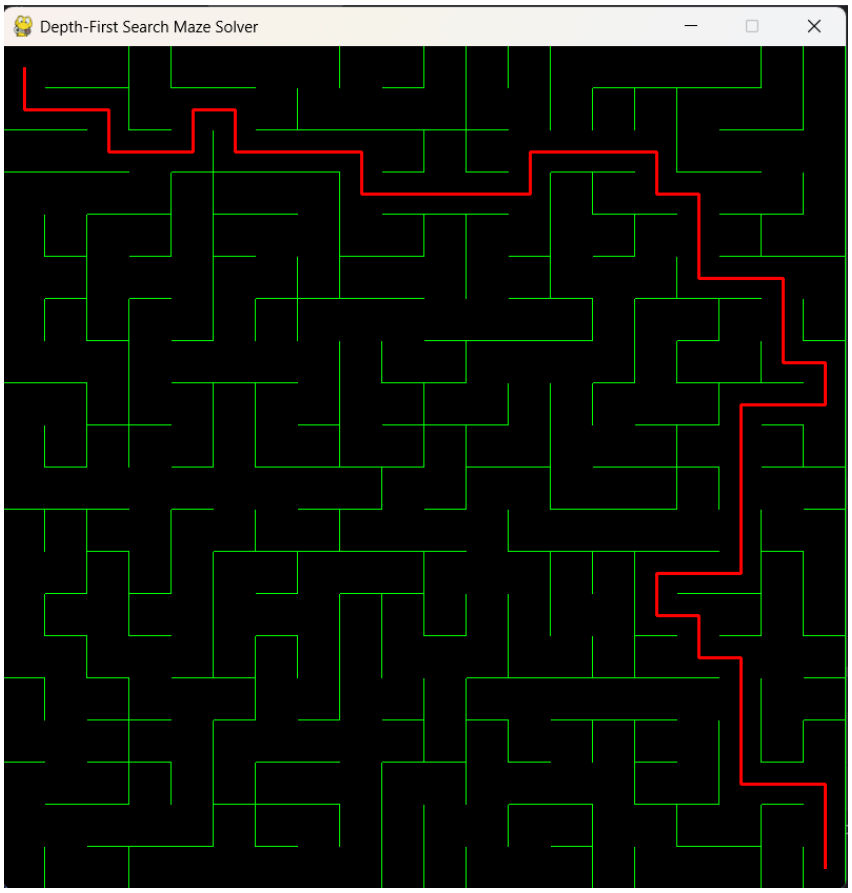

Figure 3.2: Maze solving with DFS

Listing 3.2: Key function for solving maze with DFS

```
def dfs(self, start, end):
    stack = [start]
    visited = set()
    path = []
```

```
    backtrack = []

    while stack:
        current = stack.pop()
        if current in visited:
            continue
        visited.add(current)
        path.append(current)

        if current == end:
            return path

        x, y = current
        neighbors = []
        if self.grid[y][x] & N and (x, y-1) not in visited:
            neighbors.append((x, y-1))
        if self.grid[y][x] & S and (x, y+1) not in visited:
            neighbors.append((x, y+1))
        if self.grid[y][x] & E and (x+1, y) not in visited:
            neighbors.append((x+1, y))
        if self.grid[y][x] & W and (x-1, y) not in visited:
            neighbors.append((x-1, y))

        if not neighbors:
            while path and not neighbors:
                backtrack.append(path.pop())
                if path:
                    x, y = path[-1]
                    neighbors = []
                    ...
        stack.extend(neighbors)
```

3.3 Breadth-First Search (BFS)

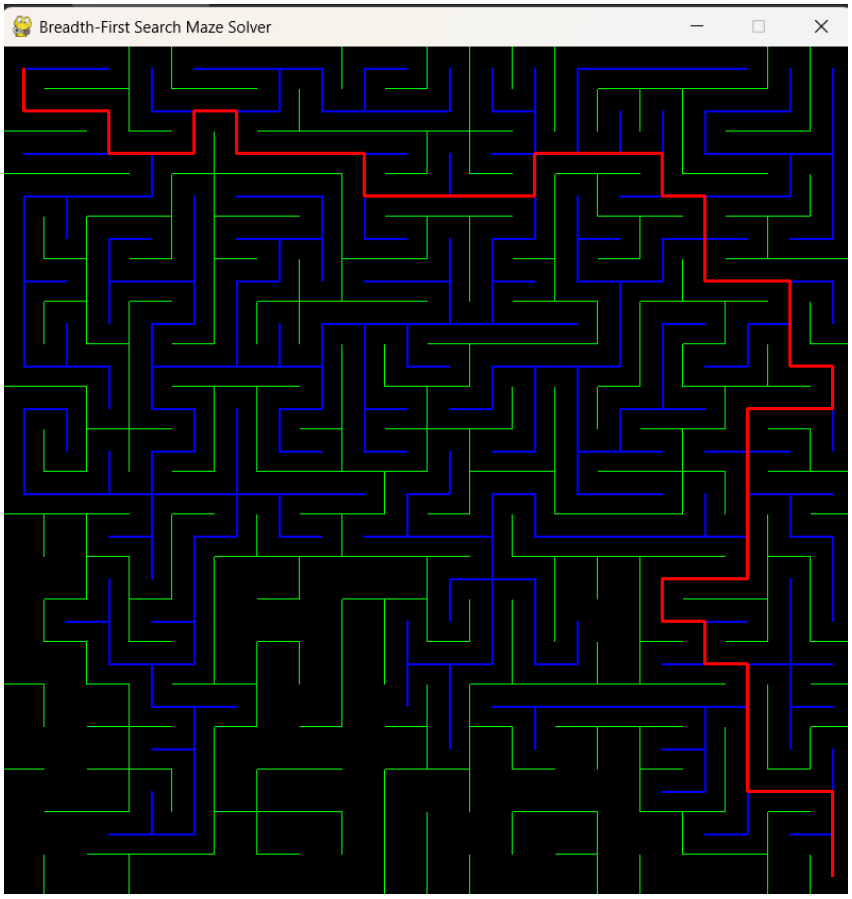

Figure 3.3: Maze solving with BFS

Listing 3.3: Key function for solving maze with BFS

```
def bfs(self, start, end):
    queue = deque([start])
    visited = set()
    parent = {start: None}
    explored = [start]

    while queue:
        current = queue.popleft()
        if current in visited:
            continue
        visited.add(current)

        if current == end:
            return self.construct_path(parent, end), explored, parent

        x, y = current
        neighbors = []
        if self.grid[y][x] & N and (x, y-1) not in visited:
            neighbors.append((x, y-1))
            parent[(x, y-1)] = current
        if self.grid[y][x] & S and (x, y+1) not in visited:
```

```
                neighbors.append((x, y+1))
                parent[(x, y+1)] = current
            if self.grid[y][x] & E and (x+1, y) not in visited:
                neighbors.append((x+1, y))
                parent[(x+1, y)] = current
            if self.grid[y][x] & W and (x-1, y) not in visited:
                neighbors.append((x-1, y))
                parent[(x-1, y)] = current

            queue.extend(neighbors)
            explored.extend(neighbors)
```

3.4 Dijkstra's Algorithm

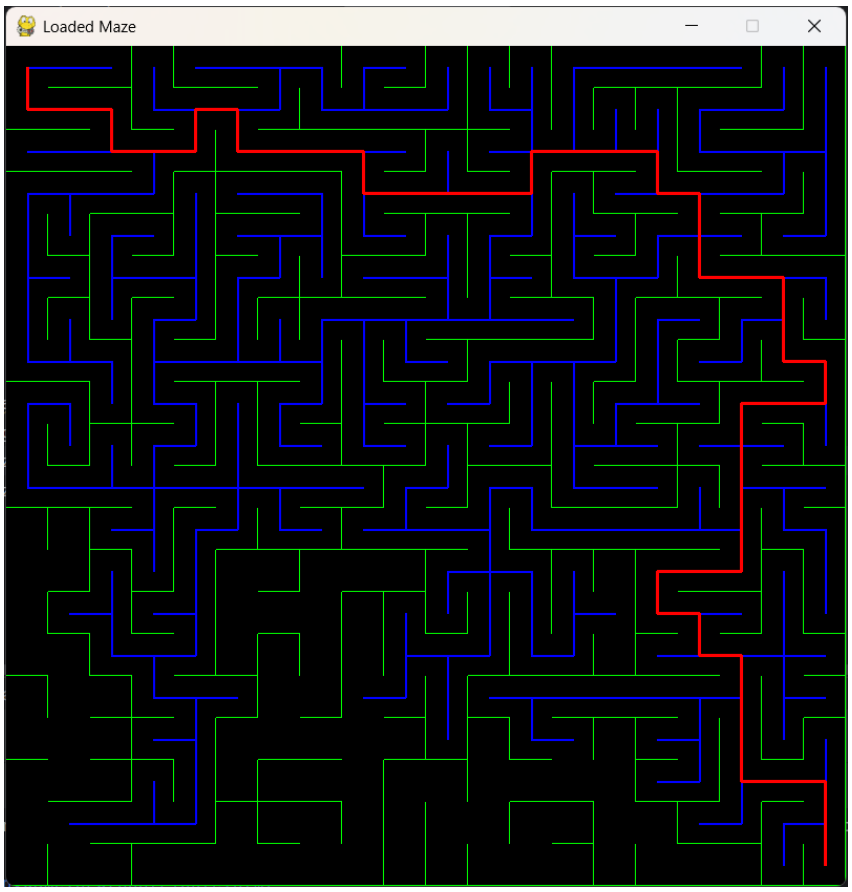

Figure 3.4: Maze solving with Dijkstra's Algorithm

Listing 3.4: Key function for solving maze with Dijkstra's Algorithm

```
def dijkstra(self, start, end):
    open_set = []
    heapq.heappush(open_set, (0, start))
    came_from = {start: None}
    g_score = {start: 0}
    explored = [start]

    while open_set:
```

```
        current = heapq.heappop(open_set)[1]
        if current == end:
            return self.construct_path(came_from, end), explored,
                came_from

        for neighbor in self.get_neighbors(current):
            tentative_g_score = g_score[current] + 1
            if neighbor not in g_score or tentative_g_score < g_score[
                neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                heapq.heappush(open_set, (g_score[neighbor], neighbor))
                explored.append(neighbor)
```

3.5 A* Algorithm

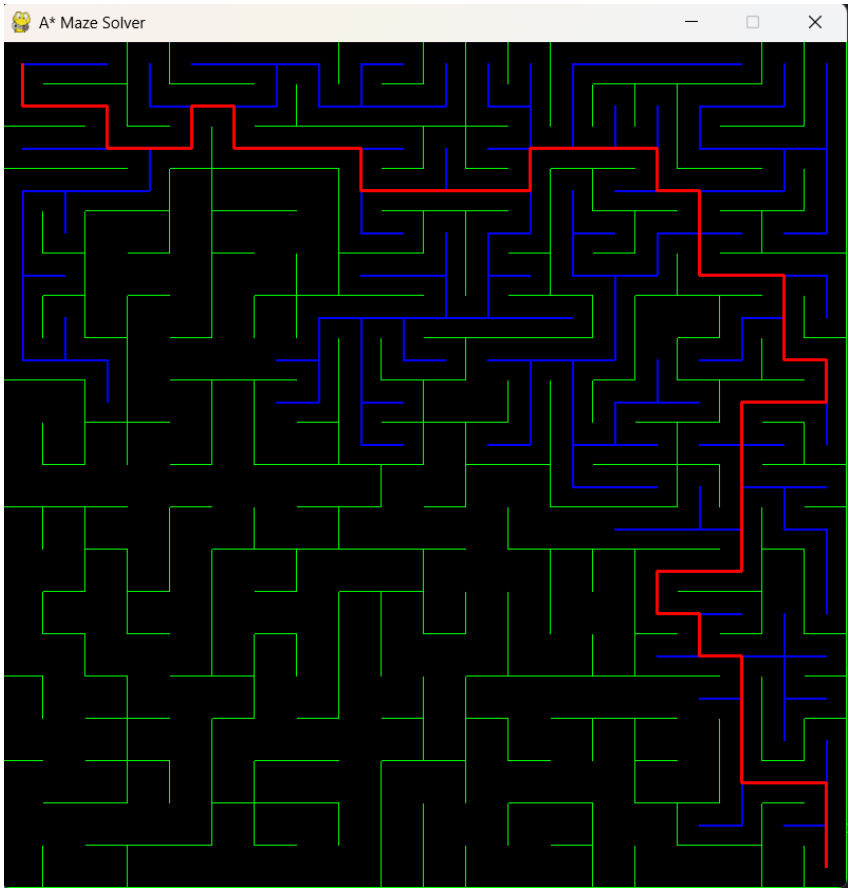

Figure 3.5: Maze solving with A* Algorithm

Listing 3.5: Key function for solving maze with A* Algorithm

```
def astar(self, start, end):
    open_set = []
    heapq.heappush(open_set, (0, start))
    came_from = {start: None}
    g_score = {start: 0}
```

```
    f_score = {start: self.heuristic(start, end)}
    explored = [start]

    while open_set:
        current = heapq.heappop(open_set)[1]
        if current == end:
            return self.construct_path(came_from, end), explored,
                came_from

        for neighbor in self.get_neighbors(current):
            tentative_g_score = g_score[current] + 1
            if neighbor not in g_score or tentative_g_score < g_score[
                neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = g_score[neighbor] + self.heuristic(
                    neighbor, end)
                heapq.heappush(open_set, (f_score[neighbor], neighbor))
                explored.append(neighbor)
```

3.6 Result Comparison Table

Algorithm	Type	Heuristic	Optimal Path	Speed	Notes
DFS	Backtracking	No	No	Medium	May revisit paths, not guaranteed to find the shortest route
BFS	Layered Search	No	Yes	High	Guarantees shortest path in unweighted graphs
Dijkstra	Cost-Based	No	Yes	High	Explores all directions equally, slower but optimal
A*	Heuristic + Cost	Yes	Yes	Medium	Guided search using heuristic, faster convergence

Table 3.1: Maze Solver Comparison

Chapter 4

Conclusion

This report demonstrated a complete pipeline for maze generation and solving using both classical and informed graph traversal algorithms. Through detailed visualizations powered by `pygame`, it became possible to not only implement but also analyze the behavioral characteristics of each approach in a step-by-step manner.

The maze was generated using **Kruskal's Algorithm**, which successfully produced a perfect maze—a connected acyclic structure with a unique path between any two nodes. The visualization of the algorithm's progression gave a clear understanding of how a spanning tree is formed using randomization and disjoint set operations.

Four traversal algorithms were then employed to solve the maze:

- **DFS** proved to be simple and fast in execution, though it lacks optimality due to its depth-oriented strategy, which can lead to long or inefficient paths.
- **BFS** provided reliable shortest paths in all cases, expanding uniformly layer by layer. The tradeoff was a larger memory footprint due to storing frontier nodes.
- **Dijkstra's Algorithm**, although more computationally intensive, systematically explored all possible paths and ensured optimality without relying on heuristics.
- **A* Algorithm** stood out as the most balanced in terms of speed and accuracy. By leveraging a heuristic (Manhattan distance), it prioritized promising directions and reduced the number of unnecessary explorations.

The visual outputs and algorithm traces presented in this report, including figures of path exploration and final routes, provided strong pedagogical value. The animations showed how algorithms traverse decision spaces differently—some aggressively, some cautiously, some with foresight.

In conclusion, each algorithm had its own merits depending on the goal—be it simplicity, completeness, optimality, or efficiency. The A* algorithm emerged as the most effective strategy for real-time maze solving where both performance and accuracy are desired. The project not only deepened understanding of search algorithms but also demonstrated the benefit of visualization in algorithmic education and analysis.