PROJECT

For

<Shortest Path Finder in a 2D Grid with Obstacles using Dijkstra's Algorithm>

By

ZAIN ALI ASIF (14875)

BSCS 3D

To

Sir Jamal Abdul Ahad

Date: 24-12-2024

DEPARTMENT OF COMPUTER SCIENCE ABBOTTABAD UNIVERSITY OF SCIENCE AND TECHNOLOGY

TABLE OF CONTENTS

1. Introduction

1.1 Purpose

This Software Requirements Specification (SRS) document describes the detailed requirements for the "Shortest Path Finder in a 2D Grid with Obstacles using Dijkstra's Algorithm." The system aims to compute the shortest path between two points in a grid environment, considering obstacles that block certain paths. The purpose of this project is to provide a reliable and efficient tool for understanding pathfinding algorithms and their practical applications in areas such as robotics, game development, and navigation systems.

1.2 Document Conventions

- **Boldface**: Used to highlight critical terms and system components.
- *Italics*: Provides additional notes, comments, or clarifications.
- • Bullets: Organizes related items in a list format for better readability.
- Numbers: Sequentially arrange steps, actions, or items based on importance or order.

1.3 Project Scope

The primary goal of this project is to develop a standalone Python application that computes the shortest path in a 2D grid using Dijkstra's algorithm. The program will:

- Support grids of various sizes, allowing users to specify dimensions.
- Identify and avoid obstacles, ensuring a feasible path is calculated.
- Display results, including a clear visual representation of the path taken.
- Enable users to input grid configurations, starting, and ending points.

The system can be used for educational purposes, research experiments, and as a foundation for more complex systems like autonomous vehicle navigation or delivery route optimization.

1.4 References

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
2. Python documentation: https://docs.python.org
3. Relevant research papers and articles on Dijkstra's algorithm and grid-based pathfinding.

2. Overall Description

2.1 Product Perspective

The Shortest Path Finder is an independent Python-based system designed for standalone use. It computes the shortest path within a 2D grid environment while considering obstacles that prevent direct traversal. The project is particularly useful for educational settings to demonstrate the functionality of Dijkstra's algorithm. Additionally, it serves as a prototype for advanced applications in real-world scenarios, such as network routing, urban planning, and autonomous systems.

2.2 User Classes and Characteristics

- **Students and Educators**: These users will utilize the system for learning and teaching purposes. The tool simplifies the understanding of Dijkstra's algorithm and grid-based navigation concepts through hands-on experimentation.
- **Software Developers**: Developers can use this system as a foundational module to integrate pathfinding capabilities into larger applications, such as simulation software or interactive games.

2.3 Operating Environment

- **Hardware Requirements**:
  - A computer system capable of running Python 3.x.
  - At least 4 GB of RAM for smooth performance.
- **Software Requirements**:
  - Python 3.x installed on the system.
  - Additional libraries such as `heapq` and `sys` included in standard Python installations.
  -

2.4 Design and Implementation Constraints

- The grid representation assumes a binary format, where obstacles are marked as `1` and open cells as `0`.
- Movement within the grid is limited to four directions: up, down, left, and right.
- The algorithm assumes that the start and goal positions are not blocked by obstacles.

2.5 Assumptions and Dependencies

- The grid provided by the user is valid and rectangular.
- Users have basic knowledge of Python to input data and interpret outputs.
- Dependencies include the Python interpreter and its standard libraries, ensuring compatibility across various platforms.

---

3. System Features

3.1 Shortest Path Computation

- **Description**: This feature implements Dijkstra's algorithm to determine the shortest path between a given starting and ending point in a grid. It calculates the optimal path by evaluating the cost of traversal across adjacent grid cells.
- **Inputs**:
  - A 2D grid where cells are marked as either `0` (open) or `1` (obstacle).
  - The coordinates of the starting point.
  - The coordinates of the destination point.
- **Outputs**:
  - A list of grid coordinates representing the shortest path.
  - If no path is found, the system outputs `None` with an appropriate message.
- **Details**:
  - The system uses a priority queue to ensure efficient computation.
  - Each cell's cost is updated based on its distance from the starting point.

3.2 Obstacle Avoidance

- **Description**: This feature ensures that the pathfinding algorithm respects obstacles within the grid. Obstacles marked as `1` are treated as impassable, and the algorithm adjusts its computation to bypass them.
- **Details**:
  - Obstacles are identified during the initial grid parsing process.
  - The algorithm dynamically recalculates paths when obstacles block direct traversal.

---

4. External Interface Requirements

4.1 User Interfaces

- The system operates via a Command Line Interface (CLI). Users input grid data, starting and destination coordinates, and receive textual outputs detailing the computed path.
- Error messages are displayed for invalid inputs, such as out-of-bound coordinates or unfeasible paths.

4.2 Hardware Interfaces

- The system does not require any specialized hardware. It runs on general-purpose computing devices with basic processing power.

4.3 Software Interfaces

- **Python Libraries**:
  - `heapq`: Used for implementing the priority queue in Dijkstra's algorithm.
  - `sys`: Facilitates efficient handling of system-level operations.

---

5. Quality Attributes

5.1 Performance

- The system is optimized for grids up to 100x100 in size. Larger grids may be processed, but performance could vary based on system specifications.
- The algorithm minimizes computational overhead by leveraging efficient data structures like priority queues.

5.2 Usability

- The CLI is designed to be intuitive, requiring minimal user training.
- Clear instructions and error messages are provided to guide users through the input process.
- Example grids and configurations are included in the documentation for reference.

5.3 Security

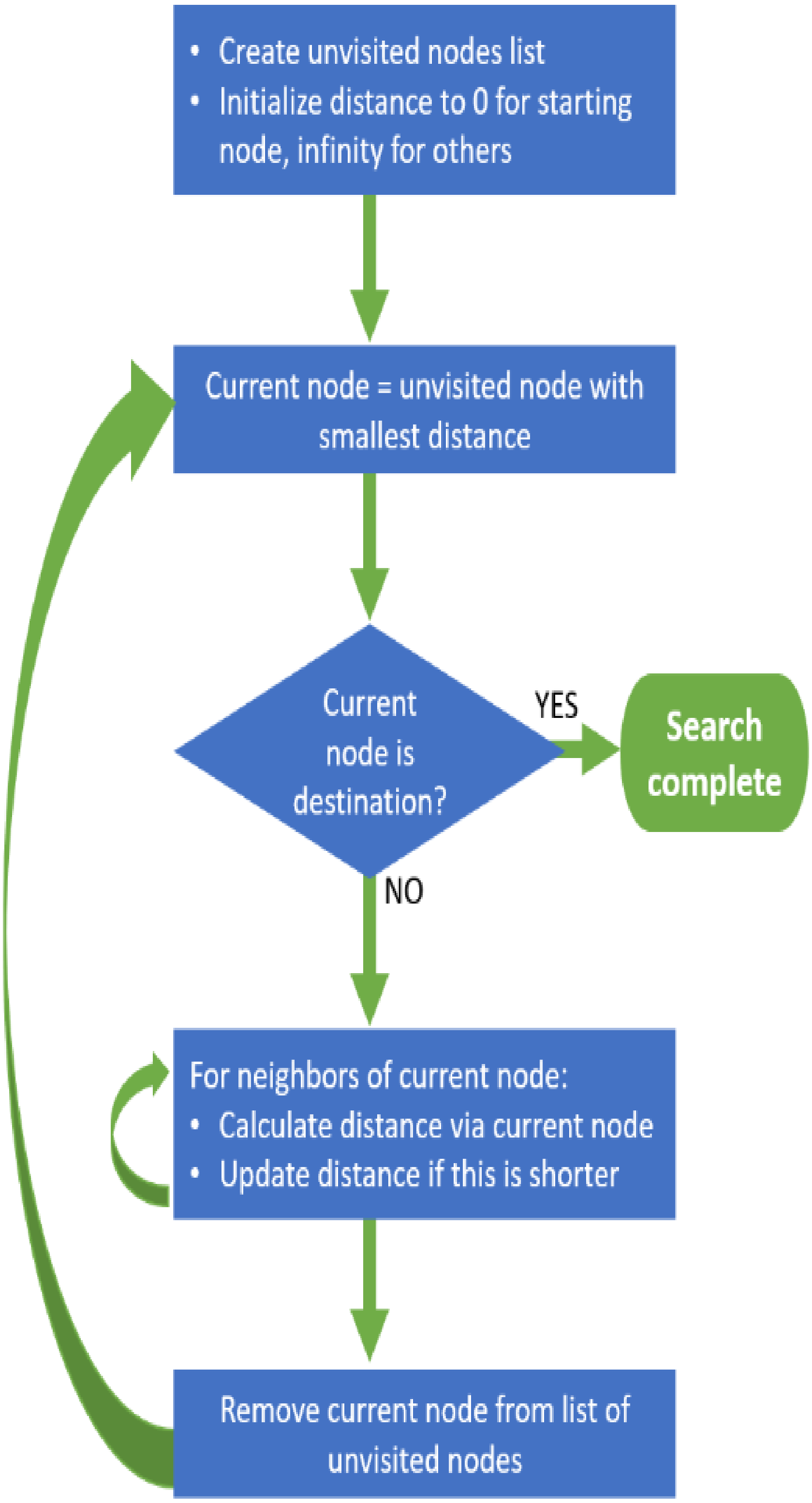
- The program processes only user-provided grid data, ensuring no sensitive or external data is involved.
- Input validation prevents potential misuse or system crashes due to invalid configurations.

Project Code Documentation

For

<Shortest Path Finder in a 2D Grid with Obstacles using Dijkstra's Algorithm>

FLOWCHART



• Create unvisited nodes list
• Initialize distance to 0 for starting node, infinity for others
Current node = unvisited node with smallest distance
Current node is destination?
YES
Search complete
NO
For neighbors of current node:
• Calculate distance via current node
• Update distance if this is shorter
Remove current node from list of unvisited nodes

CODE

```
import heapq

def dijkstra(grid, start, goal):
    """
    Finds the shortest path in a 2D grid using Dijkstra's algorithm.
    :param grid: 2D list representing the grid (0 for open cell, 1 for
obstacle)
    :param start: Tuple (x, y) for starting position
    :param goal: Tuple (x, y) for goal position
    :return: Shortest path as a list of coordinates or None if no path
exists
    """
    rows, cols = len(grid), len(grid[0])
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left,
Up

    # Priority queue for Dijkstra's algorithm
    pq = [(0, start)]  # (cost, (x, y))
    visited = set()
    came_from = {}

    while pq:
        cost, current = heapq.heappop(pq)

        if current in visited:
            continue

        visited.add(current)

        # Check if we reached the goal
        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1]  # Reverse path

        # Explore neighbors
        for dx, dy in directions:
            neighbor = (current[0] + dx, current[1] + dy)
            if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols:  #
Check bounds
                if grid[neighbor[0]][neighbor[1]] == 0 and neighbor not in
visited:  # Not an obstacle
                    heapq.heappush(pq, (cost + 1, neighbor))
                    if neighbor not in came_from:
                        came_from[neighbor] = current

    return None  # No path found
```

```
# Example usage
grid = [
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [1, 1, 0, 1, 0],
    [0, 0, 0, 0, 0],
]

start = (0, 0)
goal = (4, 4)

path = dijkstra(grid, start, goal)
if path:
    print("Shortest path:", path)
else:
    print("No path found.")
```

Output

Output   Clear

```
Shortest path: [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (4, 2),
    (4, 3), (4, 4)]


=== Code Execution Successful ===
```