INDEX

Abstract

The Smart Maze Navigator is a maze-based navigation tool designed to visualize the principles of Dijkstra's algorithm in a structured environment. Developed using Python and the pyamaze library, the system allows users to explore shortest-path algorithms in a customizable grid-based maze. With the capability to place hurdles that add traversal costs, users can observe how obstacles impact pathfinding and cost optimization. Primarily aimed at students, educators, and developers, the Smart Maze Navigator provides an interactive and educational platform for learning about graph traversal and shortest-path calculations. The system features configurable maze dimensions, real-time visualization, and intuitive controls, making it accessible and adaptable for various learning and experimentation needs. This project not only showcases algorithmic efficiency but also emphasizes the real-world application of pathfinding strategies, serving as a foundation for further exploration in fields like robotics, logistics, and artificial intelligence

.

Introduction

The **Shortest Path Problem** is a fundamental concept in graph theory and computer science, where the objective is to determine the shortest path between two nodes (or vertices) in a graph. One of the most widely-used algorithms to solve this problem is **Dijkstra's Algorithm**. Developed by Edsger W. Dijkstra in 1956, this algorithm efficiently finds the shortest path from a single source node to all other nodes in a weighted graph, where the weights represent the cost or distance between the nodes.

In this documentation, we explore the implementation of Dijkstra's Algorithm to find the shortest path between two specific nodes in a graph. Dijkstra's Algorithm is optimal for graphs that do not contain negative weight edges and is known for its efficiency, operating in **O(V + E log V)** time complexity with the use of priority queues, where **V** is the number of vertices and **E** is the number of edges.

The **Dijkstra Algorithm** is applied across various real-world scenarios, such as in GPS navigation systems to determine the fastest route, network routing protocols to optimize data transmission paths, and in many optimization problems in logistics, robotics, and AI.

The Dijkstra's Algorithm Maze Solver is a project that demonstrates the application of Dijkstra's Algorithm in solving the shortest path problem within a maze. Dijkstra's Algorithm, a well-known algorithm in computer science, is primarily used for finding the shortest paths between nodes in a graph, which can represent a variety of structures, including road networks, communication networks, and mazes.

In this project, the maze is represented as a grid of cells, where each cell is a node in the graph, and the walls between cells represent the absence of edges. The algorithm starts at a designated source point (usually the bottom-right corner) and aims to find the shortest path to a target point (typically the top-left corner), while navigating around obstacles and taking into account any additional movement costs.

Objective

The main objectives of this project are:

1. **To Implement Dijkstra's Algorithm**:
    - Develop an efficient program that uses Dijkstra's Algorithm to find the shortest path between two points in a maze, taking into account any hurdles or obstacles.
2. **To Handle Complex Mazes with Variable Costs**:
    - Incorporate agents that represent hurdles with specific movement costs, allowing the system to simulate real-world challenges where different paths have varying difficulties.
3. **To Visualize the Shortest Path**:
    - Provide a clear visual representation of the maze, agents, and the shortest path found by the algorithm. This helps users understand how the algorithm navigates through the maze and how it responds to obstacles.
4. **To Load and Process Predefined Maze Structures**:
    - Support the loading of maze configurations from external CSV files, allowing for easy customization and testing of different maze scenarios.
5. **To Enhance Understanding of Pathfinding Algorithms**:
    - Through interactive visualizations and real-time path computation, demonstrate the mechanics of Dijkstra's Algorithm and its application in finding the shortest paths in graph-based problems.
6. **To Optimize Performance for Larger Mazes**:
    - Ensure the algorithm performs efficiently, even for larger mazes with multiple agents and hurdles, by leveraging priority queues and optimal data structures.

Technology & Tool used

i. **Programming Language:** Python

- Python was selected for its simplicity, extensive library support, and ability to rapidly develop and test algorithms. It also provides a rich ecosystem of libraries for visualization and algorithm implementation.

ii. Libraries/Packages Used:

- pyamaze:
    - A Python package used for creating and visualizing mazes, agents, and paths.
    - This library simplifies the process of maze generation and provides intuitive visual representations of the maze and pathfinding algorithms.
    - Installed via pip install pyamaze.
- csv:
    - Used to load pre-defined maze configurations from a CSV file (djkMaze.csv).
    - This helps in loading and testing different maze structures with ease.
    - Part of Python's standard library.

iii. Integrated Development Environment (IDE):

- VS Code / PyCharm / Jupyter Notebook:
  - Any modern code editor or IDE can be used for writing and testing the Python code. These editors provide syntax highlighting, code completion, and integrated debugging, making the development process easier.

iv. Visualization Tools:

- pyamaze provides a graphical representation of mazes and agents, making it easier to visualize the steps of Dijkstra's Algorithm and the path traced through the maze.
- textLabel is used within pyamaze to display additional information, like the total cost of the shortest path.

v. Operating System:

- The project can be developed and run on Windows, macOS, or Linux platforms. Python's cross-platform compatibility ensures that the project runs smoothly across various operating systems

Requirements

- **Software Requirements**
  1. **Python 3.x**:
     - Primary programming language for implementing Dijkstra's algorithm and managing maze configurations.
     - Downloadable from Python Official Website.
  2. **Python Libraries**:
     - **pyamaze**: For maze creation, grid visualization, and path tracing.
     - **Other Libraries**: Basic Python libraries like math and time if needed for auxiliary functions and testing.
  3. **Text Editor/IDE**:
     - **VS Code**, **PyCharm**, or any other Python IDE for coding, testing, and debugging.
  4. **Operating System**:
     - Compatible with Windows, macOS, and Linux, as long as Python 3.x is supported.

- **Hardware Requirements**
  1. **Processor**:
     - Standard multi-core processor (Intel i3/AMD Ryzen 3 or above) for smooth processing, especially with larger maze configurations.
  2. **Memory (RAM)**:
     - Minimum 4GB RAM; 8GB or more recommended for handling larger grids and efficient memory management.
  3. **Storage**:
     - Minimal storage requirement (~100MB) to accommodate Python installation and project files.
  4. **Display**:
     - Resolution of at least 1280x720 for clear visualization of the maze and pathfinding progress.
  5. **Input Devices**:
     - Mouse and keyboard for interaction with the interface and maze configuration.

     These requirements ensure the **Smart Maze Navigator** runs smoothly on most modern personal computers, providing the necessary computational power and display capabilities for efficient pathfinding and visualization.

Existing System

In the realm of maze navigation and pathfinding, several existing systems and applications demonstrate the principles of algorithms like Dijkstra's, A*, and Breadth-First Search. However, many of these applications either lack user interaction, fail to include configurable traversal costs, or are limited to predefined setups that do not allow for user customization. Here's an overview of common existing systems and their limitations in the context of educational and interactive use:

1. **Basic Maze Solvers**:
   - Numerous basic pathfinding demos exist, often limited to a fixed grid size with predefined paths and obstacles.
   - These systems typically solve for the shortest path but lack flexibility in terms of obstacle configuration, dynamic hurdle costs, and adjustable pathfinding parameters.

Need for the Smart Maze Navigator

The **Smart Maze Navigator** was developed to address these gaps by:

- Allowing full user interaction with the maze layout, including the placement and cost adjustment of hurdles.
- Using a clear visualization that displays each step of Dijkstra's algorithm, enhancing understanding.
- Offering flexibility to adjust the maze layout and obstacle configurations, making it a more engaging tool for educational and research applications

Proposed System

The **Smart Maze Navigator** introduces an interactive and user-friendly platform to visualize and experiment with Dijkstra's algorithm in a customizable maze environment. This system is designed to offer flexibility, educational value, and hands-on experience with pathfinding concepts, addressing the limitations of existing tools.

Key Features of the Proposed System

1. **Customizable Maze Generation**:
   - The system generates a grid-based maze, which users can customize by selecting dimensions or loading predefined configurations.
   - A dynamic approach allows users to visualize the maze structure, setting specific start and end points.
2. **Dijkstra's Algorithm with Adjustable Hurdles**:
   - The Smart Maze Navigator implements Dijkstra's algorithm to find the shortest path from the starting point to the end point.
   - Users can add hurdles with adjustable traversal costs, which directly influence the pathfinding process, making the system ideal for understanding how different obstacles impact shortest-path calculations.
3. **Real-Time Visualization**:
   - The system provides real-time feedback on the path chosen by the algorithm, marking each step visually within the maze.
   - Path cells, start and end points, and hurdles are color-coded for clarity, allowing users to observe Dijkstra's algorithm's decision-making process interactively.

System Design

The **Smart Maze Navigator** system design focuses on modular architecture, allowing flexibility in maze generation, algorithm execution, and user interaction. Below are the major design components that together form a cohesive and interactive pathfinding tool using Dijkstra's algorithm.

1. Architecture Overview

The system is built around a modular structure with three main components:

- **Maze Generation and Configuration**: Generates and customizes the maze structure.
- **Pathfinding and Algorithm Execution**: Executes Dijkstra's algorithm to find the shortest path, considering obstacle costs.
- **Visualization and User Interface**: Provides real-time feedback and allows user interaction with the maze and algorithm.

2. Component Design

1. **Maze Generation and Configuration**
    - **Maze Grid**: The maze is represented as a grid of cells, each with four potential walls in directions (north, south, east, west). The grid layout defines pathways, walls, start, and end points.
    - **Obstacle Placement**: Users can place hurdles with varying traversal costs at specific cells. Each hurdle cell has a weight representing its additional cost to traverse.
    - **Configuration Options**: Users can generate a maze with preset dimensions, load predefined mazes, or create custom setups, making the maze highly adaptable to various pathfinding scenarios.
2. **Pathfinding and Algorithm Execution**
    - **Dijkstra's Algorithm**:
        - Uses a dictionary (unvisited) to store tentative distances for each cell, starting with infinity for all cells except the starting cell.
        - Iterates over cells, continually selecting the cell with the shortest distance from the start and updating neighboring cells' costs based on hurdle weights.
    - **Reverse Path Calculation**: Tracks each cell's predecessor to construct the shortest path once the algorithm reaches the end cell.
    - **Handling Hurdles**: Checks for hurdle cells during execution and adds the respective traversal cost to the distance calculation dynamically.
3. **Visualization and User Interface**
    - **Real-Time Path Visualization**: The selected shortest path is visually highlighted in the maze, with color codes for the start, end, path, and hurdles.
    - **User Feedback**: A display label shows the total path cost, helping users understand how each obstacle affects the overall distance.
    - **Interactive Controls**: Users can start/stop the algorithm, add/remove obstacles, or change configurations to rerun pathfinding in real time.

3. Data Flow and Control Flow

- **Data Flow**:
    - Maze layout and obstacle configurations are stored as grid data, allowing easy access for both the visualization and pathfinding components.
    - During execution, the algorithm continuously updates distance and predecessor data, storing it for both pathfinding calculations and visualization updates.

- **Control Flow**:
    - The process starts with the maze generation/configuration phase, where users define the maze layout and hurdles.
    - When users trigger the algorithm, control passes to Dijkstra's function, which processes cells iteratively until it reaches the end.
    - After completing the path calculation, control is passed to the visualization module to display the calculated shortest path.

4. Technology Stack

- **Python**: Primary language for algorithmic processing and maze configuration.
- **pyamaze Library**: Facilitates maze visualization, agent path tracing, and user interaction.
- **Standard Libraries**: Additional Python libraries like math for auxiliary functions, as needed.

5. User Interface Design

The user interface has intuitive controls that allow:

- **Maze Configuration**: Options to select maze size, place start/end points, and add/remove hurdles with different costs.
- **Algorithm Execution**: Buttons to start, pause, or reset Dijkstra's algorithm.
- **Real-Time Feedback**: Display labels and color-coded cells for immediate visualization of pathfinding progress and total traversal cost.

Source Code & Output

```
from pyamaze import maze, agent, COLOR, textLabel
def dijstra(m,*h):

    hurdles = [(i.position,i.cost) for i in h]

    unvisited = {n: float('inf') for n in m.grid}
    unvisited[(m.rows, m.cols)] = 0
    visited = {}
    revPath = {}

    while unvisited:
        currCell = min(unvisited, key=unvisited.get)
        visited[currCell] = unvisited[currCell]
        if currCell == (1, 1):
            break
        for d in 'EWNS':
            if m.maze_map[currCell][d]:
                if d == 'E':
                    childCell = (currCell[0], currCell[1] + 1)
                elif d == 'W':
                    childCell = (currCell[0], currCell[1] - 1)
                elif d == 'S':
```

```
                childCell = (currCell[0] + 1, currCell[1])
            elif d == 'N':
                childCell = (currCell[0] - 1, currCell[1])
            if childCell in visited:
                continue
            tempDist = unvisited[currCell] + 1
            for hurdle in hurdles:
                if hurdle[0] == currCell:
                    tempDist += hurdle[1]

            if tempDist < unvisited[childCell]:
                unvisited[childCell] = tempDist
                revPath[childCell] = currCell
        unvisited.pop(currCell)

    fwdPath = {}
    cell = (1, 1)
    while cell!=(m.rows, m.cols):
        fwdPath[revPath[cell]] = cell
        cell = revPath[cell]

    return fwdPath, visited[(1, 1)]


if __name__ == '__main__':
    myMaze = maze(6, 6)
    # myMaze.CreateMaze(loopPercent=100)
    myMaze.CreateMaze(loadMaze='djkMaze.csv')

    h1 = agent(myMaze, 4, 6, color=COLOR.red)
    # h2 = agent(myMaze, 4, 6, color=COLOR.red)
    h2 = agent(myMaze, 4, 4, color=COLOR.red)

    h1.cost = 100
    # h2.cost = 100
    h2.cost = 100

    path, c = dijstra(myMaze, h1, h2)
    textLabel(myMaze, 'Total Cost', c)

    a = agent(myMaze, color=COLOR.cyan, filled=True, footprints=True)
    myMaze.tracePath({a: path})

    myMaze.run()
```

Output


PYTHON MAZE WORLD by Learning Orbis
Total Cost : 10


PYTHON MAZE WORLD by Learning Orbis
Total Cost : 14

Implementation Details

The **Smart Maze Navigator** leverages Python and the pyamaze library to build a user-friendly maze navigation tool using Dijkstra's algorithm. This section outlines the key implementation steps and components.

1. Maze Generation and Setup

The pyamaze library was used to create a 6x6 maze grid with designated start and end points. The maze was either generated randomly or loaded from a pre-defined file (djkMaze.csv). Each cell in the maze has walls in four directions—north, south, east, and west—that define its accessibility to neighboring cells.

2. Agent and Hurdle Configuration

Agents representing hurdles were added to the maze, each with a specified position and additional traversal cost. These hurdles act as obstacles, increasing the pathfinding cost for cells they occupy. The cost attribute for each hurdle is configurable, which allows users to experiment with different hurdle placements and effects on pathfinding.

3. Implementation of Dijkstra's Algorithm

The core of the project is the dijstra function, which calculates the shortest path while accounting for hurdle costs.

- **Data Structures**:
  - unvisited: Holds cells with their tentative distances from the starting cell.
  - visited: Stores cells that have been processed, along with their final shortest distance.
  - revPath: Maps each cell to its predecessor to reconstruct the path.
- **Path Calculation**:
  - The algorithm iteratively selects the cell with the shortest tentative distance, updates its neighbors, and tracks the reverse path to the destination. Hurdle costs are added dynamically based on the presence of hurdles in the path.

4. User Interface and Labels

Text labels were added to display the total path cost dynamically, enhancing the user experience by providing immediate feedback on the pathfinding results.

System Study

The **Smart Maze Navigator** project was thoroughly studied to define system requirements, identify user needs, and ensure alignment with the intended functionalities. The following outlines the system study, encompassing user roles, functional requirements, and technical specifications.

1. Purpose and Scope

The purpose of the **Smart Maze Navigator** is to provide a tool that visually demonstrates pathfinding algorithms, specifically Dijkstra's algorithm, in a grid-based maze. By introducing hurdles with different traversal costs, users can better understand how obstacles impact pathfinding efficiency. The system is designed for educational and experimental use, allowing students, educators, and enthusiasts to explore algorithmic navigation in a structured environment.

2. User Roles and Needs

- **Students and Learners**: To understand and observe the working of pathfinding algorithms and analyze how traversal costs impact path selection.
- **Educators**: To demonstrate the principles of Dijkstra's algorithm and shortest-path strategies in real-time, using an intuitive interface.
- **Developers/Researchers**: To experiment with modifications or alternative algorithms, as well as scalability and performance in varying maze sizes.

3. Functional Requirements

- **Maze Generation**:
  - Ability to generate a maze of configurable size.
  - Option to load predefined maze configurations to test specific scenarios.

- **Pathfinding with Dijkstra's Algorithm**:
    - Implement Dijkstra's algorithm to calculate the shortest path from the maze's start to end.
    - Dynamically adjust pathfinding calculations based on hurdles with varying traversal costs.

4. Technical Specifications

- **Programming Language**: Python 3.x, chosen for its simplicity and the availability of necessary libraries.
- **Libraries and Tools**:
    - **pyamaze**: For maze creation and visualization, allowing easy manipulation of maze grids and path tracing.
    - **User Interface Elements**: Simple text labels and color codes for displaying paths, start/end points, and hurdles.
- **System Requirements**:
    - Runs on standard desktop systems without needing high-performance computing resources.
    - Requires installation of Python and the pyamaze library, both lightweight and easy to set up.

6. Study Outcome

The system study affirmed the feasibility and educational value of the **Smart Maze Navigator**. With its modular design, intuitive user interface, and focus on Dijkstra's algorithm, the project effectively meets the needs of its target users. The system study further guided the project's design and testing phases, ensuring that all functional and non-functional requirements align with user needs and educational objectives.

Technical Feasibility

The **Smart Maze Navigator** project was evaluated for technical feasibility to assess its practicality, resources, and alignment with project goals. Here are the key areas considered:

1. Technology Stack

- **Programming Language**: Python was chosen for its readability, extensive library support, and ease of integration with visualization tools. Python's simplicity also makes it accessible for educational purposes and prototyping.
- **Library Support**: The project utilizes pyamaze, a Python library tailored for maze creation and visualization, which simplified the generation of grid-based mazes and the addition of agents. Pyamaze's built-in functions allowed for rapid development of the maze visualization and agent positioning, reducing time spent on interface design.
- **Algorithm Implementation**: Dijkstra's algorithm, a well-known graph traversal algorithm, was implemented to handle the maze navigation with additional costs for obstacles. The algorithm is efficient for the maze size used in this project, making it feasible without advanced hardware or excessive processing power.

2. Resource Requirements

- **Hardware**: The project runs smoothly on standard desktop or laptop systems with modest CPU and memory requirements. No specialized hardware is required, as the maze size and Dijkstra's algorithm can be handled by average computational resources.
- **Software and Tools**: The project relies only on Python and the pyamaze library, both of which are open-source and freely available. This minimal setup makes it feasible without extra financial cost for software licenses or proprietary tools.

3. Scalability and Future Enhancements

- **Scalability**: While effective within small to medium-sized mazes, the project could face scalability challenges with significantly larger or dynamic mazes. Dijkstra's algorithm, though efficient for smaller grids, could become slower in larger setups without optimizations.

System Testing

System testing for the **Smart Maze Navigator** aimed to ensure that the entire application functions as expected, with all integrated components working seamlessly. Here is a breakdown of the key testing areas and outcomes:

1. Functional Testing

Tested each primary function to confirm that all features work according to specifications:

- **Pathfinding Accuracy**: Verified that Dijkstra's algorithm consistently identifies the shortest path from the start to the end cell, taking hurdle costs into account.
- **Hurdle Handling**: Ensured that hurdles are correctly processed with additional costs and that these influence pathfinding.
- **Maze Generation and Display**: Confirmed that the maze is generated according to set dimensions and displayed accurately, with start and end points clearly marked.

2. Usability Testing

Assessed the user experience to ensure that the system is intuitive:

- **Interface Clarity**: Verified that the maze, hurdles, and paths are visually distinct, with color-coded paths and agents that enhance readability.
- **User Interaction**: Tested the ease of modifying hurdles, costs, and initiating pathfinding, ensuring that users can quickly adjust configurations without errors.

3. Performance Testing

Evaluated the application's responsiveness and speed under different conditions:

- **Maze Size Variability**: Tested with different maze sizes to check for any slowdowns, ensuring that the algorithm performs efficiently with increased maze complexity.
- **Concurrent Actions**: Simulated multiple rapid user actions (e.g., adjusting hurdles and re-running the algorithm) to ensure the system remains stable without lags.

4. Boundary Testing

Checked how the system handles edge cases:

- **Single-Path Mazes**: Verified that Dijkstra's algorithm finds paths in simpler, straightforward mazes without dead-ends.
- **Obstacle-Dense Mazes**: Ensured the algorithm correctly navigates and finds alternative paths in heavily obstructed mazes.

5. Error Handling and Recovery

Confirmed that the system can handle and recover from unexpected inputs or configurations:

- **Invalid Maze Configurations**: Tested the response when no valid path is available, ensuring the system alerts the user appropriately.
- **Nonexistent Start/End Path**: Assured that the system correctly identifies when a path is impossible due to obstacles.

Conclusion

The **Smart Maze Navigator** project successfully demonstrates the application of Dijkstra's algorithm to solve complex maze navigation problems while incorporating obstacles and varied traversal costs. By combining algorithmic rigor with a user-friendly interface, the project not only highlights the efficiency of pathfinding techniques but also offers an engaging educational tool for learning about algorithms.
The project's modular design and use of the pyamaze library enabled a streamlined implementation, allowing for easy visualization and customization of maze paths, start and end points, and obstacle configurations. Comprehensive testing ensured functionality, reliability, and responsiveness across various maze configurations, enhancing user interaction and real-time feedback.
Looking forward, the **Smart Maze Navigator** could be extended to include additional algorithms, real-time obstacle adjustments, and mobile compatibility. This project demonstrates the potential of algorithmic pathfinding in fields such as robotics, gaming, and educational technology, offering a foundation for further exploration and development.

Reference

- **Online Resources**:
  - GeeksforGeeks. (n.d.). Dijkstra's Shortest Path Algorithm Using Python. Retrieved from GeeksforGeeks
    - This article offers a practical implementation of Dijkstra's algorithm in Python, serving as a reference for understanding the algorithm's structure and logic.
- **Books and Articles**:
  - Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
    - This book provides a comprehensive overview of algorithms, including Dijkstra's algorithm, its applications, and performance analysis.
- **Development Tools**:
  - Python Software Foundation. (n.d.). Python. Retrieved from Python Official Website
    - The official Python website provides resources for downloading Python, as well as extensive documentation for learning and implementing Python programming.