Project Report On

Path Finding Visualizer

Submitted by

Prittam Bhattacharyya - 26371021004
Sayan Bhattacharjee - 26371021009
Bhagyasri Mazumder - 26371021016
Debangan Roy - 26371021020
Gourab Malakar - 26371021048

Under the guidance of

Mr. Arup Mallick
Dept of Master of Computer Applications
Regent Education and Research Foundation

Department of ***Master of Computer Applications***

Regent Education and Research Foundation

(Affiliated to Maulana Abul Kalam Azad University of Technology, West Bengal)

Barrackpore - 700121, Barrackpore, WB

Regent Education and Research Foundation

(Affiliated to Maulana Abul Kalam Azad University of Technology, West Bengal)

Barrackpore - 700121, Barrackpore, WB

Certificate of Approval

This is to certify that this report of MCA Final Year project, entitled **“Path Finding Visualizer”** is a record of bona-fide work, carried out by **Prittam Bhattacharyya, Sayan Bhattacharjee, Bhagyasri Mazumder, Debangan Roy, Gourab Malakar** under my supervision and guidance.

In my opinion, the report in its present form is in partial fulfillment of all the requirements, as specified by the ***Regent Education and Research Foundation*** and as per regulations of the ***Maulana Abul Kalam Azad University of Technology***. In fact, it has attained the standard, necessary for submission. To the best of my knowledge, the results embodied in this report, are original in nature and worthy of incorporation in the present version of the report for PG programme in Master of Computer Applications in the year 2022-2023.

Guide / Supervisor

Mr. Arup Mallick
Department of Master of Computer Applications
Regent Education and Research Foundation

Examiner(s)

Head of the Department

Master of Computer Applications
Regent Education and Research Foundation

ACKNOWLEDGEMENT

Success of any project depends largely on the encouragement and guidelines of many others. I take this sincere opportunity to express my gratitude to the people who have been instrumental in the successful completion of this project work.

I would like to show our greatest appreciation to **Mr. Arup Mallick**, Professor, Dept. of Master of Computer Application, Regent Education and Research Foundation. I always feel motivated and encouraged every time by his valuable advice and constant inspiration; without his encouragement and guidance this project would not have materialized. Words are inadequate in offering our thanks to the other trainees, project assistants and other members for their encouragement and cooperation in carrying out this project work. The guidance and support received from all the members and who are contributing to this project, was vital for the success of this project.

Words are inadequate in offering our thanks to the other trainees, project assistants and other members for their encouragement and cooperation in carrying out this project work. The guidance and support received from all the members and who are contributing to this project, was vital for the success of this project.

Mr. Prittam Bhattacharyya
Registration No: 212630571010003
Roll No: 26371021004

Mr. Sayan Bhattacharjee
Registration No: 212630571010020
Roll No: 26371021013

Ms. Bhagyasri Mazumder
Registration No: 212630571010026
Roll No: 26371021016

Mr. Debangan Roy
Registration No: 212630571010012
Roll No: 26371021020

Mr. Gourab Malakar
Registration No: 212630571010045
Roll No: 26371021048

PROJECT SYNOPSIS

In today's digital age, where efficiency and optimization are paramount, finding the most effective route from point A to point B is a ubiquitous challenge. Whether it's navigating a complex maze, planning a logistics network, or simulating autonomous vehicle paths, the ability to visualize and analyze the best path has become a crucial endeavor.

This is where the Path Finding Visualizer steps in - a powerful tool that brings clarity and innovation to the realm of navigation.

Path Finding Visualizer is a valuable tool for understanding and analyzing pathfinding algorithms. By visualizing these algorithms in action, we can gain insights into their mechanisms, compare their performance, and explore their strengths and weaknesses.

This visualizer provides an interactive platform to study the intricate processes of

- Dijkstra's algorithm
- Breadth First Search (BFS)
- Depth First Search (DFS)

After loading in the project, we will first see a set of controls at the top and a blank grid at the bottom of the page. The top part consists of all the 3 algorithms and buttons to generate mazes and to clear out the grid. We can either draw walls in the grid or generate a random maze then set the starting point and ending point. After that we can select an algorithm and click on visualize to see the algorithm in action.

CONTENTS

1. INTRODUCTION

Welcome to the technical document for the Path Finding Visualizer, a powerful tool designed to demonstrate and analyze various pathfinding algorithms. This document aims to provide a comprehensive overview of the features, functionality, and underlying technology of the Path Finding Visualizer, allowing developers and users to gain a deeper understanding of its capabilities.

Pathfinding is a fundamental problem in the field of computer science and has extensive applications in areas such as robotics, gaming, logistics, and network routing. The Path Finding Visualizer serves as an educational and practical tool that allows users to visualize the process of finding optimal paths in different scenarios.

The main objective of this document is to provide a detailed explanation of the Path Finding Visualizer's architecture, its core components, and the algorithms it employs. Additionally, we will explore the interactive user interface and discuss the available customization options to tailor the visualization to specific needs.

Whether you are a developer interested in extending the functionalities of the Path Finding Visualizer or an enthusiast looking to learn more about pathfinding algorithms, this document will serve as a valuable resource. By the end, you will have a thorough understanding of the tool's inner workings and be equipped to utilize it effectively in your projects.

Let us now delve into the fascinating world of pathfinding algorithms and explore the capabilities and intricacies of the Path Finding Visualizer.

1.1 Objective

The objective of this technical document is to provide a comprehensive understanding of the Path Finding Visualizer, a powerful tool for visualizing and analyzing pathfinding algorithms. By the end of this document, readers should have a clear understanding of the following:

- Architecture and Components: Gain insights into the overall architecture of the Path Finding Visualizer, including its core components and their interactions. Understand how the tool is structured to facilitate efficient pathfinding visualization.

- Pathfinding Algorithms: Explore various pathfinding algorithms implemented in the Path Finding Visualizer, such as Dijkstra's algorithm, BFS and DFS. Understand the principles behind these algorithms and their applications in different scenarios.

- Visualization and Interaction: Learn about the interactive user interface of the Path Finding Visualizer and how to utilize its features effectively. Discover customization options to adapt the visualization to specific requirements and explore ways to interact with the visualization in real-time.

- Practical Applications: Understand the practical applications of pathfinding algorithms and how the Path Finding Visualizer can be leveraged in different domains, such as game development, robotics, logistics, and network routing. Explore real-world use cases and scenarios where pathfinding algorithms play a crucial role.

- Extension and Integration: For developers, learn how to extend the functionalities of the Path Finding Visualizer by integrating additional algorithms or features. Understand the tools and resources available to modify and enhance the tool according to specific project requirements.

By achieving these objectives, readers will have a solid foundation to utilize the Path Finding Visualizer effectively, gain a deeper understanding of pathfinding algorithms, and apply this knowledge to their own projects, research, or educational endeavors.

1.2 Scope of the System

The scope of the Path Finding Visualizer system encompasses the following key aspects:

- **Pathfinding Visualization**: The system provides a visual representation of pathfinding algorithms, allowing users to observe the step-by-step process of finding optimal paths in a graph or grid-based environment. The visualization includes highlighting visited nodes, explored paths, and the final optimal path.

- **Algorithm Selection**: The system supports multiple pathfinding algorithms, including but not limited to Dijkstra's algorithm, A* algorithm, Breadth-First Search (BFS), and Depth-First Search (DFS). Users can select and compare different algorithms to understand their strengths, weaknesses, and performance characteristics.

- **Grid Customization**: The system allows users to create custom grids or import predefined grid layouts. Users can define start and target positions, as well as place obstacles or barriers to simulate real-world scenarios. This feature enables the visualization of pathfinding algorithms in various environments and problem spaces.

- **Real-time Interaction**: Users can interact with the visualization in real-time, pausing, resuming, and resetting the algorithm execution. Additionally, they can modify the grid during the visualization process by adding or removing obstacles, altering start and target positions, and observing the algorithm's response to these changes.

- **Performance Analysis**: The system provides performance metrics for each executed algorithm, such as the total distance traveled, number of visited nodes, and execution time. This analysis allows users to compare and evaluate the efficiency and effectiveness of different algorithms in specific scenarios.

- **Educational Resources**: The system offers educational resources, including tooltips, algorithm descriptions, and explanations of the pathfinding process. These resources aim to enhance the understanding of pathfinding algorithms and their applications, making the system suitable for educational purposes.

- Extensibility: The system is designed with extensibility in mind, allowing developers to add new pathfinding algorithms or customize existing ones. This flexibility enables the integration of cutting-edge algorithms or tailored solutions to meet specific project requirements.

It is important to note that while the Path Finding Visualizer provides a powerful platform for visualizing and analyzing pathfinding algorithms, it does not encompass broader functionalities such as multi-agent pathfinding, dynamic environments, or advanced optimization techniques. The system focuses primarily on visualizing individual pathfinding algorithms in a static grid environment.

1.3 Feasibility Study

A feasibility study is conducted to assess the viability and practicality of a project. In the case of the Path Finding Visualizer, the following aspects have been evaluated to determine its feasibility:

- **Technical Feasibility**: The Path Finding Visualizer relies on existing technologies and algorithms for pathfinding. These algorithms have been extensively studied and implemented, demonstrating their technical feasibility. The system utilizes well-established programming languages and frameworks commonly used for web development, ensuring that the necessary technical infrastructure is readily available.

- **Resource Feasibility**: The resources required for the Path Finding Visualizer, such as hardware, software, and human resources, are readily accessible. It can be developed and deployed on standard computer systems with adequate processing power and memory. The required software tools and frameworks are widely used and available, minimizing resource constraints. Additionally, the development team possesses the necessary skills and expertise to design and implement the system effectively.

- **Time Feasibility**: The development timeline and the estimated time required to complete the Path Finding Visualizer have been considered. The project can be executed within a reasonable timeframe, considering the complexity of the system and the availability of the development team. Adequate planning and project management methodologies can be employed to ensure timely completion.

- **Financial Feasibility**: The financial implications of developing and maintaining the Path Finding Visualizer have been evaluated. The costs associated with hardware, software licenses, development tools, and maintenance can be reasonably managed within the allocated budget. Additionally, the potential benefits and value derived from the system, such as its educational and practical applications, can outweigh the associated costs.

- **User Acceptance**: The feasibility study also considers the potential user acceptance and demand for the Path Finding Visualizer. Given the widespread interest in pathfinding algorithms, particularly in fields such as computer science education, game development, and robotics, there is a substantial target audience for the system. User feedback and engagement can be actively sought to continuously improve and enhance the system's usability and functionality.

Based on the evaluation of these feasibility factors, the Path Finding Visualizer project demonstrates strong feasibility in terms of technical aspects, resource availability, development timeline, financial considerations, and potential user acceptance. This study concludes that the development and implementation of the Path Finding Visualizer are viable and justifiable, paving the way for the successful creation of the system.

1.3.1 Technical Feasibility

The technical feasibility of the Path Finding Visualizer project is a crucial aspect that determines whether the required technology and infrastructure are available to develop and deploy the system. The following factors contribute to the technical feasibility:

- Algorithms and Libraries: Pathfinding algorithms, such as Dijkstra's algorithm and A*, have been extensively studied and implemented in various programming languages. There are numerous open-source libraries and code resources available that provide efficient implementations of these algorithms. This ensures that the necessary algorithms can be readily incorporated into the Path Finding Visualizer.

- Programming Languages and Frameworks: The choice of programming languages and frameworks plays a significant role in the technical feasibility of the project. The development of the Path Finding Visualizer can be carried out using widely adopted languages such as JavaScript, Python, or Java. These languages have robust ecosystems and extensive libraries that support web development and data visualization.

- Web Development Technologies: The Path Finding Visualizer is primarily a web-based application, leveraging the capabilities of modern web browsers. Web development technologies such as HTML5, CSS, and JavaScript provide the necessary tools to create interactive user interfaces, handle user input, and render visual representations of pathfinding algorithms. These technologies are well-established and widely supported, ensuring compatibility across different platforms and devices.

- Performance Considerations: Pathfinding algorithms can be computationally intensive, especially when dealing with large grids or complex graphs. However, advancements in hardware capabilities and optimization techniques enable the efficient execution of pathfinding algorithms within reasonable time frames. Additionally, the system can incorporate optimizations such as heuristics or pruning techniques to enhance performance without compromising the accuracy of the visualization.

- Scalability and Extensibility: The Path Finding Visualizer can be designed to accommodate future expansions and enhancements. The system architecture can be modular and flexible, allowing the integration of additional algorithms or customization options. This extensibility ensures that the system can adapt to evolving needs and incorporate emerging pathfinding techniques or user requirements.

- Compatibility and Accessibility: The web-based nature of the Path Finding Visualizer enables accessibility across multiple platforms and devices. It can be accessed through popular web browsers, making it compatible with different operating systems. The system can be designed to be responsive, ensuring optimal user experience on various screen sizes and resolutions.

Based on the availability of well-established algorithms, programming languages, frameworks, and web development technologies, as well as the considerations for performance, scalability, and compatibility, the technical feasibility of the Path Finding Visualizer project is high. The necessary tools and resources are readily accessible, empowering developers to create a robust and efficient system for pathfinding visualization.

1.3.2 Operational Feasibility

Operational feasibility assesses the practicality and effectiveness of implementing and operating the Path Finding Visualizer. The following factors contribute to the operational feasibility of the project:

- User Requirements: Understanding the needs and requirements of the target users is crucial for operational feasibility. Conducting user research and gathering feedback from potential users, such as developers, students, or researchers, ensures that the Path Finding Visualizer aligns with their expectations and provides value. User engagement throughout the development process helps tailor the system to their specific needs.

- User Interface and Usability: The user interface (UI) of the Path Finding Visualizer should be intuitive, user-friendly, and visually appealing. A well-designed UI facilitates ease of interaction, making it simple for users to understand and utilize the system's features. Incorporating usability testing and iterative design processes ensures that the UI is optimized for efficient user engagement.

- Training and Support: Providing adequate training and support resources is essential for operational feasibility. Documentation, tutorials, and online guides can assist users in understanding the system's functionalities and utilizing them effectively. Additionally, offering responsive customer support channels, such as forums or email support, helps address user queries and troubleshoot issues promptly.

- Maintenance and Updates: The Path Finding Visualizer may require periodic updates and maintenance to address bugs, enhance performance, and introduce new features. Planning for regular maintenance and ensuring a streamlined update process is crucial for sustaining the system's operational feasibility. Establishing version control systems and bug tracking mechanisms aids in efficient maintenance and updates.

- Scalability and Performance: Considering the potential growth in user base and increasing demand, the system should be designed to handle scalability and maintain optimal performance. Implementing efficient algorithms, optimizing code, and utilizing appropriate data structures contribute to the system's ability to handle a larger number of users and maintain acceptable response times.

- Data Management and Security: The Path Finding Visualizer may involve storing user data, such as saved grids or user preferences. Implementing proper data management practices and ensuring data security measures, such as encryption and secure authentication, protect user information and contribute to operational feasibility.

- Integration and Compatibility: Ensuring compatibility with various browsers, operating systems, and devices is important to maximize the system's reach and accessibility. Conducting compatibility tests and providing clear system requirements help users understand the necessary environment for optimal system operation.

By addressing user requirements, focusing on user interface design and usability, providing training and support, planning for maintenance and updates, considering scalability and performance, implementing proper data management and security practices, and ensuring integration and compatibility, the operational feasibility of the Path Finding Visualizer is enhanced. This comprehensive approach ensures that the system can be effectively implemented, operated, and maintained to meet the needs of its users.

1.3.3 Economic Feasibility

Economic feasibility examines the financial viability and cost-effectiveness of the Path Finding Visualizer project. The following factors contribute to the economic feasibility of the project:

- Cost-Benefit Analysis: A cost-benefit analysis is conducted to evaluate the potential benefits derived from the Path Finding Visualizer against the costs associated with its development, implementation, and maintenance. The benefits can include educational value, increased efficiency in algorithm understanding, and potential contributions to research and development in pathfinding algorithms. Comparing these benefits with the incurred costs helps determine the economic feasibility of the project.

- Development Costs: The costs involved in developing the Path Finding Visualizer include human resources, software licenses, hardware, and development tools. These costs can be estimated by considering the skillset and experience of the development team, the duration of the project, and the necessary development environment. The economic feasibility is assessed by comparing the development costs with the potential benefits and value derived from the system.

- Maintenance and Upkeep Costs: The ongoing costs associated with maintaining and updating the Path Finding Visualizer should be considered. This includes the costs of bug fixes, feature enhancements, server hosting, and customer support. Planning for regular maintenance and allocating resources for updates and support is crucial for assessing the long-term economic feasibility of the project.

- Revenue Generation or Cost Recovery: Economic feasibility can be further assessed by exploring revenue generation or cost recovery opportunities. The Path Finding Visualizer may have potential revenue streams, such as offering premium features, licensing the software to educational institutions or organizations, or incorporating advertisements or sponsorships. Identifying these opportunities helps offset the development and maintenance costs and contributes to the economic viability of the project.

- Return on Investment (ROI): Evaluating the potential ROI of the Path Finding Visualizer project is important for economic feasibility. ROI considers the financial gains and benefits derived from the system in relation to the invested resources. This assessment helps stakeholders determine the value and profitability of the project and make informed decisions regarding its implementation and sustainability.

- Cost Reduction and Optimization: Identifying opportunities to reduce costs and optimize resources can enhance the economic feasibility of the project. This includes exploring open-source alternatives for development tools, leveraging cloud-based infrastructure for hosting, and optimizing code and algorithms for improved performance and efficiency. These measures help minimize expenses while maintaining the desired system functionality.

By conducting a thorough cost-benefit analysis, considering development and maintenance costs, exploring revenue generation

2. SOFTWARE REQUIREMENT SPECIFICATION (SRS)

1. Introduction

1.1 Purpose

The purpose of this Software Requirement Specification (SRS) is to provide a detailed description of the requirements for the development of the Path Finding Visualizer.

1.2 Scope

The scope of this project includes the development of a web-based Path Finding Visualizer that allows users to interactively visualize and analyze various pathfinding algorithms.

1.3 Software Used

The software used for the development of the Path Finding Visualizer includes:

- Visual Studio Code (VS Code): A lightweight, open-source code editor that provides a rich set of features for efficient code development and editing.
- Google Chrome: A widely used web browser that supports modern web technologies and provides a platform for testing and running the web-based Path Finding Visualizer.

2. Functional Requirements

2.1 User Interface

- The system shall provide an intuitive and user-friendly interface for users to interact with the Path Finding Visualizer.
- The user interface shall include options to create or import custom grids, select algorithms, and visualize the pathfinding process.

2.2 Pathfinding Algorithms

- The system shall support multiple pathfinding algorithms, including Dijkstra's algorithm, A* algorithm, Breadth-First Search (BFS), and Depth-First Search (DFS).
- Users shall be able to select and visualize different algorithms and observe their behavior in finding optimal paths.

2.3 Grid Customization

- The system shall allow users to create custom grids or import predefined grid layouts.
- Users shall be able to define start and target positions and place obstacles or barriers in the grid to simulate different scenarios.

2.4 Real-time Interaction

- Users shall have the ability to pause, resume, and reset the algorithm execution in real-time.
- Users shall be able to modify the grid during the visualization process by adding or removing obstacles and altering start and target positions.

3. Non-functional Requirements

3.1 Performance

- The system shall execute pathfinding algorithms efficiently, providing real-time visualization even for large grids or complex scenarios.
- The system shall aim to minimize execution time and provide responsive user interactions.

3.2 Compatibility

- The system shall be compatible with Google Chrome, ensuring consistent behavior across different versions and platforms.
- The system shall be responsive and compatible with various screen sizes and resolutions.

3.3 Security

- The system shall incorporate appropriate security measures to protect user data and prevent unauthorized access.
- User authentication and data encryption shall be implemented where necessary.

4. Constraints

- The development of the Path Finding Visualizer shall be carried out using Visual Studio Code as the primary code editor.
- The web-based system shall be tested and validated for compatibility and functionality on Google Chrome as the primary browser.

5. References

- The requirements for the Path Finding Visualizer are based on user feedback, domain knowledge, and best practices in pathfinding algorithm visualization.

Note: This Software Requirement Specification (SRS) provides an overview of the key requirements for the Path Finding Visualizer. Further details, such as specific user interface design, algorithms, and performance metrics, may be documented in separate design and test specifications.

3. SOFTWARE DEVELOPMENT PROCESS MODEL

The software development process model adopted for the development of the Path Finding Visualizer is an iterative and incremental model. This model allows for flexibility, adaptability, and continuous improvement throughout the development lifecycle. The following phases were followed:

1. **Requirements Gathering**: The initial phase involved gathering and analyzing user requirements for the Path Finding Visualizer. This included understanding the desired features, functionality, and user expectations for the system.

2. **Design:** In this phase, the system architecture, user interface design, and data structures were defined. The design took into consideration the usability, scalability, and extensibility of the Path Finding Visualizer. The React component-based architecture was chosen to ensure modularity and reusability.

3. **Development:** The development phase involved writing the code for the Path Finding Visualizer using React with Vite and TypeScript. React was chosen as the JavaScript library for building the user interface, while Vite provided a fast development environment with hot module replacement. TypeScript was utilized to introduce type safety and enhance code quality. Tailwind CSS, a utility-first CSS framework, was used for efficient and responsive styling.

4. **Testing:** Testing was conducted at multiple levels to ensure the quality and correctness of the software. Unit tests were performed to validate individual components and algorithms. Integration tests were carried out to verify the interaction between different modules. User acceptance testing (UAT) was conducted to gather feedback and validate the system's functionality and usability.

5. **Deployment:** The Path Finding Visualizer was deployed to a web server for users to access and utilize. The deployment process involved configuring the necessary infrastructure, such as server hosting and domain setup, to make the system available to users.

6. **Maintenance and Updates**: After deployment, the system required ongoing maintenance and periodic updates. Bug fixes, performance optimizations, and feature enhancements were implemented based on user feedback and identified requirements. The iterative nature of the development process facilitated continuous improvement and ensured that the Path Finding Visualizer remained up-to-date and efficient.

Throughout the development process, agile principles such as regular communication, collaboration, and adaptation were embraced. The use of an iterative and incremental development model allowed for continuous feedback and incorporation of changes, resulting in a robust and user-focused Path Finding Visualizer.

4. OVERVIEW

The Path Finding Visualizer is a web-based application that provides an interactive platform to visualize and analyze various pathfinding algorithms. The project aims to assist users in understanding the behavior and efficiency of different algorithms in finding optimal paths within a grid-based environment.

Key Features:
1. Grid: The Path Finding Visualizer presents a grid structure where users can define the start and target positions. The grid represents the environment where the pathfinding algorithms operate.

2. **Algorithm Selection:** Users have the option to choose from different pathfinding algorithms, including Dijkstra's algorithm, Breadth-First Search (BFS), and Depth-First Search (DFS). This selection enables users to observe and compare the behaviors and outcomes of these algorithms.

3. **Maze Generation:** The system offers the ability to generate random or recursive mazes within the grid. These mazes can introduce obstacles and barriers, providing a more challenging scenario for the pathfinding algorithms to navigate.

4. **Grid and Path Manipulation:** Users can interact with the grid by adding or removing obstacles, altering the start and target positions, and clearing the grid or path when necessary. This flexibility allows users to experiment with different scenarios and observe the algorithm's behavior accordingly.

5. **Performance Metrics:** The Path Finding Visualizer provides valuable insights into the performance of the algorithms. It displays metrics such as the time taken to find the path, the number of cells traveled by the algorithm, and the number of cells scanned or explored during the search process. These metrics assist users in evaluating the efficiency and effectiveness of different algorithms.

The Path Finding Visualizer project is developed using React with Vite, utilizing TypeScript for type safety and Tailwind CSS for efficient styling. The iterative and incremental software development process ensures continuous improvement, incorporating user feedback and addressing identified requirements.

With its grid structure, algorithm selection, maze generation, grid and path manipulation options, and informative performance metrics, the Path Finding Visualizer provides a user-friendly and educational tool for visualizing and comparing various pathfinding algorithms.

4.1 System Overview

The Path Finding Visualizer is a web-based application that allows users to interactively visualize and analyze different pathfinding algorithms. It provides a user-friendly interface and a grid-based environment where users can define start and target positions, create obstacles, and observe how algorithms navigate through the grid to find the optimal path.

The system comprises several components that work together to deliver a seamless user experience:

1. **User Interface:** The user interface presents a grid structure where users can define the start and target positions and create obstacles within the grid. It offers intuitive controls and options for algorithm selection, maze generation, and grid manipulation.

2. **Pathfinding Algorithms:** The system supports various pathfinding algorithms, including Dijkstra's algorithm, Breadth-First Search (BFS), and Depth-First Search (DFS). Users can choose a specific algorithm and observe its execution in real-time as it searches for the optimal path.

3. **Maze Generation:** The system provides functionality for generating random or recursive mazes within the grid. These mazes introduce obstacles and barriers that pose challenges for the pathfinding algorithms, enabling users to explore their behavior in complex scenarios.

4. **Grid Manipulation:** Users can interact with the grid by adding or removing obstacles, changing the start and target positions, and clearing the grid or path. This flexibility allows for experimentation and the creation of different pathfinding scenarios.

5. **Performance Metrics:** The system captures performance metrics related to the pathfinding process. It provides information such as the time taken to find the path, the number of cells traveled by the algorithm, and the number of cells scanned or explored during the search. These metrics help users evaluate the efficiency and effectiveness of different algorithms.

6. **Visualization and Feedback:** The system offers real-time visualization of the pathfinding process, allowing users to observe how the algorithm explores the grid and finds the optimal path. It provides visual cues, such as highlighting cells or paths, to aid in understanding the algorithm's progress. Users can also provide feedback and interact with the system during the visualization process.

The Path Finding Visualizer is developed using React with Vite, incorporating TypeScript for enhanced code quality and type safety. Tailwind CSS is utilized for responsive and efficient styling. The system follows an iterative and incremental development approach to continuously improve and enhance user experience based on feedback and requirements.

Overall, the Path Finding Visualizer provides a comprehensive and interactive platform for users to explore and compare different pathfinding algorithms, visualize their behaviors, and gain insights into their performance within a grid-based environment.

4.1.1 Limitation of Existing System

1. Limited Algorithm Support: The current version of the Path Finding Visualizer has a limited set of supported algorithms, including Dijkstra's algorithm, Breadth-First Search (BFS), and Depth-First Search (DFS). While these algorithms are widely used and provide valuable insights, other advanced algorithms such as A* (A-star), Greedy Best-First Search, or Bidirectional algorithms are not supported in the current version. However, it is planned to

expand the system's capabilities in the future by adding more algorithms to provide users with a wider range of options and comparisons.

2. **Grid Size Limitations**: The size of the grid in the Path Finding Visualizer is limited by the capabilities and performance of the web browser. Extremely large grids may cause performance issues or slow down the visualization process. Users may experience decreased responsiveness or slower execution when working with grids that exceed the recommended size.

3. **Simplified Environment:** The Path Finding Visualizer simulates a simplified environment with a grid and obstacles. It does not take into account real-world complexities, such as varying terrain or dynamic obstacles. The system focuses on illustrating the behavior of the selected algorithms within a controlled environment and may not accurately represent real-life scenarios.

4. **Limited Path Customization:** While users can define the start and target positions and create obstacles within the grid, the system does not provide advanced options for specifying constraints or customizing the pathfinding behavior. The algorithms operate based on predefined rules and heuristics without user-defined constraints or optimizations.

5. **Performance Variations:** The performance of the Path Finding Visualizer may vary depending on the user's device, browser, and grid size. Older browsers or devices with limited processing power may experience reduced performance or slower execution, impacting the real-time visualization experience.

6. **Browser Compatibility:** The system is primarily developed and tested on Google Chrome, and while efforts have been made to ensure compatibility with other modern web browsers, there may be variations in behavior or display across different browser environments. Users are encouraged to use the recommended browser for the best experience.

It is important to note that these limitations are based on the current version of the Path Finding Visualizer, and future updates and enhancements are expected to address these limitations and introduce additional features, algorithms, and optimizations to improve the system's functionality and usability.

4.2 Proposed System

The proposed system aims to enhance the functionality and usability of the Path Finding Visualizer by incorporating the following features and improvements:

1. **Expanded Algorithm Support:** The proposed system will include an expanded set of pathfinding algorithms, such as A* (A-star), Greedy Best-First Search, and Bidirectional algorithms. By incorporating these advanced algorithms, users will have a broader range of options to explore and compare different pathfinding strategies.

2**. Customization Options:** The proposed system will introduce additional customization options to allow users to define specific constraints and preferences for the pathfinding algorithms. This could include factors such as weights on grid cells, allowing users to create scenarios with varying terrain or obstacles of different difficulties.

3. **Realistic Environments**: To simulate real-world scenarios, the proposed system will include more realistic environments with dynamic obstacles, varying terrains, or multiple targets. This will enable users to observe how the algorithms adapt and navigate in complex and dynamic environments.

4. **Enhanced Visualization:** The proposed system will provide improved visualizations to enhance the understanding of the pathfinding process. This could include animations, step-by-step explanations, and highlighting of key algorithmic steps. These visual aids will assist users in comprehending the algorithms' behavior and decision-making process.

5. **Performance Optimization**: The proposed system will focus on optimizing performance to handle larger grids and more complex scenarios efficiently. This could involve algorithmic optimizations, caching techniques, and leveraging parallel processing capabilities to ensure smooth and responsive real-time visualizations.

6. **User Interface Enhancements**: The proposed system will feature an improved user interface design with intuitive controls, better organization of options and settings, and responsive layouts. The user interface will prioritize ease of use, accessibility, and seamless interaction to provide a user-friendly experience.

7. **Collaboration and Sharing**: The proposed system may include features for collaboration and sharing. Users will have the ability to save and share their grid configurations, algorithms, and visualizations with others. This will facilitate knowledge sharing, collaborative learning, and enable users to showcase their work or seek feedback from the community.

8. **Mobile and Tablet Support:** The proposed system will be optimized for mobile and tablet devices, ensuring a responsive and user-friendly experience across various screen sizes. This will allow users to access and utilize the Path Finding Visualizer on a wide range of devices.

The proposed system aims to enhance the capabilities of the Path Finding Visualizer, providing users with a more comprehensive and customizable platform for exploring and understanding pathfinding algorithms. By incorporating additional algorithms, customization options, realistic environments, improved visualization, and performance optimizations, the proposed system will offer a powerful tool for educational, research, and problem-solving purposes.

4.2.1 Objectives of the proposed system

1. Expand Algorithm Support: The primary objective of the proposed system is to incorporate a wider range of pathfinding algorithms, including advanced algorithms such as A* (A-star), Greedy Best-First Search, and Bidirectional algorithms. This objective aims to provide users with a comprehensive selection of algorithms to explore and compare, enabling them to gain a deeper understanding of different pathfinding strategies.

2. **Enhance Customization Options**: The proposed system seeks to introduce additional customization options, allowing users to define specific constraints and preferences for the pathfinding algorithms. Users will have the ability to customize factors such as weights on grid cells, enabling the creation of more complex scenarios with varying terrains and obstacles of different difficulties. This objective aims to provide users with a greater level of control and flexibility in designing and analyzing pathfinding scenarios.

3. **Simulate Realistic Environments**: The proposed system aims to simulate more realistic environments within the pathfinding visualizer. This includes incorporating dynamic obstacles, varying terrains, or multiple targets. By introducing these elements, users will be able to observe and analyze how the algorithms adapt and navigate in complex and dynamic scenarios. This objective aims to provide a more realistic and practical learning experience for users.

4. **Improve Visualization and Explanations:** The proposed system seeks to enhance the visualization of the pathfinding process, providing improved animations, step-by-step explanations, and highlighting of key algorithmic steps. This objective aims to enhance users' understanding of the algorithms' behavior and decision-making process, making the visualizer more educational and informative.

5. **Optimize Performance**: The proposed system focuses on optimizing performance to handle larger grids and more complex scenarios efficiently. By implementing algorithmic optimizations, caching techniques, and leveraging parallel processing capabilities, the system aims to ensure smooth and responsive real-time visualizations. This objective aims to provide users with a seamless and enjoyable user experience, even in demanding pathfinding scenarios.

6. **Enhance User Interface and Experience**: The proposed system aims to improve the user interface design, with intuitive controls, better organization of options and settings, and responsive layouts. This objective focuses on enhancing the overall user experience by prioritizing ease of use, accessibility, and seamless interaction with the pathfinding visualizer.

7. **Enable Collaboration and Sharing**: The proposed system may include features for collaboration and sharing. Users will have the ability to save and share their grid configurations, algorithms, and visualizations with others. This objective aims to foster knowledge sharing, collaborative learning, and community engagement within the pathfinding visualizer.

8. **Ensure Mobile and Tablet Support**: The proposed system will be optimized for mobile and tablet devices, ensuring a responsive and user-friendly experience across various screen sizes. This objective aims to provide users with the flexibility to access and utilize the pathfinding visualizer on a wide range of devices, enhancing accessibility and usability.

Overall, the objectives of the proposed system are to enrich the functionality, customization options, realism, visualization, performance, and user experience of the pathfinding visualizer. By addressing these objectives, the system aims to provide a powerful and versatile tool for exploring, analyzing, and learning about pathfinding algorithms.

4.2.2 Users of the Proposed system

1. **Students and Learners:** The proposed system is designed to be an educational tool for students and learners studying computer science, algorithms, or artificial intelligence. It provides a hands-on platform for understanding and experimenting with various pathfinding algorithms. Students can use the system to visualize and compare different algorithms, observe their behaviors, and gain insights into their efficiency and effectiveness.

2. **Researchers and Developers**: Researchers and developers in the field of algorithms and artificial intelligence can benefit from the proposed system. It offers a platform to test, analyze, and benchmark different pathfinding algorithms. Researchers can use the system to validate their own algorithms or compare them with existing ones. Developers can utilize the system as a reference or learning resource when implementing pathfinding functionality in their applications.

3. **Educators and Instructors:** The proposed system can be a valuable tool for educators and instructors teaching courses related to algorithms, data structures, or artificial intelligence. It provides a visual and interactive learning experience for students, helping them grasp the concepts of pathfinding algorithms more effectively. Educators can use the system to demonstrate algorithmic concepts, engage students in hands-on activities, and facilitate class discussions on algorithmic behaviors and optimizations.

4. **Problem Solvers and Analysts:** Professionals and enthusiasts engaged in problem-solving activities, such as logistics optimization, route planning, or game development, can utilize the proposed system as a tool for analyzing and visualizing pathfinding scenarios. It provides a practical platform to evaluate different algorithms' performances and make informed decisions based on the observed results. The system can aid in optimizing routes, identifying bottlenecks, or designing efficient pathfinding strategies in real-world applications.

5. **Enthusiasts and Hobbyists:** Pathfinding algorithms and visualizations can be of interest to hobbyists and enthusiasts who have a passion for algorithms, puzzles, or game development. The proposed system can serve as an engaging platform for them to explore and experiment with different algorithms, create challenging scenarios, and share their findings with the community. It offers an opportunity for enthusiasts to deepen their understanding of algorithms and contribute to the pathfinding domain.

6. **Collaborative and Online Learning Communities:** The proposed system can foster collaboration and engagement within online learning communities and forums. Users can share their grid configurations, algorithms, and visualizations with others, facilitating discussions, feedback, and collaborative learning. It can serve as a hub for knowledge sharing and community engagement, allowing users to learn from each other's experiences and collectively improve their understanding of pathfinding algorithms.

In summary, the proposed system caters to a diverse user base, including students, researchers, developers, educators, problem solvers, enthusiasts, and online learning communities. It offers a versatile platform for learning, researching, experimenting, and collaborating in the field of pathfinding algorithms.

5. ASSUMPTION AND DEPENDENCIES

1. **Assumptions:**
   a. Users have a basic understanding of pathfinding algorithms and their underlying principles.
   b. The proposed system will be accessed through modern web browsers with JavaScript support.
   c. Users have a stable internet connection to access and utilize the system without interruptions.
   d. The system assumes that the hardware and software requirements of the web browser are met for optimal performance.
   e. Users are responsible for ensuring the accuracy and correctness of the input data, such as defining start and target positions and creating obstacles within the grid.

2. **Dependencies:**
   a. Software Dependencies: The proposed system relies on various software components and libraries, including React, Vite, TypeScript, and Tailwind CSS. These dependencies need to be properly installed and configured for the system to function correctly.
   b. Browser Compatibility: The system's performance and functionality may depend on the compatibility and capabilities of the web browser being used. The system will be primarily developed and tested on Google Chrome, but efforts will be made to ensure compatibility with other modern web browsers.
   c. Development Resources: The successful implementation of the proposed system is dependent on the availability of development resources, including skilled developers, testing environments, and development tools.
   d. User Feedback and Collaboration: To enhance and improve the system, user feedback and collaboration are crucial. The system relies on user engagement and active participation to gather insights, identify issues, and implement enhancements. User feedback will be considered for future updates and improvements to the system.

These assumptions and dependencies are essential considerations for the successful development, deployment, and utilization of the proposed system. They help set realistic expectations and guide the planning and implementation process.

6. TECNOLOGIES

1. **React:** The proposed system is built using React, a popular JavaScript library for building user interfaces. React provides a component-based architecture and efficient rendering capabilities, making it suitable for creating interactive and responsive web applications.

2. **Vite:** Vite is a fast, modern build tool for web applications. It is used in the development of the proposed system to provide a seamless development experience, fast module hot-reloading, and optimized production builds. Vite enables quick iteration and enhances the overall performance of the system.

3. **TypeScript:** TypeScript is a typed superset of JavaScript that adds static typing and additional language features. It is utilized in the proposed system to enhance code maintainability, catch potential errors during development, and provide better code documentation and editor tooling support.

4. **Tailwind CSS:** Tailwind CSS is a utility-first CSS framework that provides a set of pre-built CSS classes for rapid UI development. It is used in the proposed system to simplify styling and ensure a consistent and responsive user interface. Tailwind CSS allows for easy customization and provides a flexible approach to designing the visual elements of the system.

5. **HTML5:** HTML5 is the latest version of the Hypertext Markup Language used for structuring web content. It provides essential elements and attributes for creating the user interface and organizing the components in the proposed system.

6. **CSS3:** CSS3 is the latest version of Cascading Style Sheets used for styling web content. It is utilized in the proposed system to define the visual presentation, layout, and animation of various elements and components.

7. **JavaScript:** JavaScript is a programming language that enables dynamic and interactive behavior on web pages. It is extensively used in the proposed system for implementing the core logic, handling user interactions, and orchestrating the pathfinding algorithms and visualizations.

8. **Web APIs:** The proposed system utilizes various web APIs, including the DOM API (Document Object Model) and Canvas API, to manipulate and render elements on the web page, handle user input, and create interactive visualizations.

These technologies form the foundation of the proposed system, providing the necessary tools, frameworks, and languages to develop a dynamic, interactive, and visually appealing pathfinding visualizer.

6.1 Tools used in Development

1. **Visual Studio Code (VS Code):** VS Code is a popular source code editor that provides a rich set of features and extensions for web development. It offers a lightweight and customizable environment with built-in support for JavaScript, TypeScript, and React. Developers can benefit from features like code autocompletion, syntax highlighting, debugging capabilities, and version control integration.

2. **Chrome DevTools**: Chrome DevTools is a set of web developer tools built directly into the Google Chrome browser. It offers a wide range of features for debugging, profiling, and analyzing web applications. DevTools allows developers to inspect and manipulate the DOM, monitor network activity, debug JavaScript code, and optimize performance for the proposed system.

3. **TypeScript Compiler**: The TypeScript compiler (tsc) is used to transpile TypeScript code into JavaScript code. It ensures type checking and compiles the code to a version that can be executed in web browsers. The compiler provides error checking, code optimizations, and compatibility checks to enhance the development process and produce reliable JavaScript code.

4. **Git:** Git is a distributed version control system used for tracking changes in source code during development. It allows developers to collaborate, manage different versions of the codebase, and revert to previous states if necessary. Git provides features like branching, merging, and remote repository management, ensuring efficient and organized code development.

5. **NPM (Node Package Manager):** NPM is a package manager for JavaScript that allows developers to install, manage, and share reusable code packages. It is used in the development of the proposed system to manage dependencies, libraries, and tools required for building and running the application.

6. **Chrome Browser:** The Google Chrome browser is widely used during the development process for testing and debugging the proposed system. It provides advanced debugging capabilities, performance profiling, and compatibility testing to ensure the system works seamlessly across different browser environments.

7. **ESLint**: ESLint is a popular JavaScript linter that helps identify and report coding errors, style inconsistencies, and potential issues in the codebase. It enforces coding standards and best practices, improving code quality and maintainability during development.

8. React Developer Tools: React Developer Tools is a browser extension that enhances the debugging and profiling capabilities specifically for React applications. It provides additional insights into the React component hierarchy, state management, and performance optimizations, enabling developers to analyze and troubleshoot React components effectively.

These tools play a crucial role in the development process, enabling efficient coding, debugging, testing, version control, and performance optimization for the proposed system.

6.2 Development Environment

The development environment for the proposed system consists of the following components:

1. **Operating System:** The development environment is compatible with various operating systems such as Windows, macOS, and Linux. Developers can choose the operating system that best suits their preferences and development setup.

2. **Text Editor or Integrated Development Environment (IDE):** Developers can utilize a text editor like Visual Studio Code (VS Code) or an IDE such as WebStorm or IntelliJ IDEA. These tools provide features like code highlighting, autocompletion, code navigation, and debugging capabilities, enhancing the development workflow.

3. **Node.js and NPM**: Node.js is a JavaScript runtime that allows developers to execute JavaScript code outside of a web browser. It provides access to the Node Package Manager (NPM), which is used for managing dependencies, libraries, and tools required for building and running the proposed system.

4. **Git and Version Control:** Git is a distributed version control system used for tracking changes in source code. Developers can use Git to manage different versions of the codebase, collaborate with team members, and revert to previous states if necessary. Online platforms like GitHub or GitLab can be utilized for hosting and managing repositories.

5. **Browser:** The proposed system is primarily developed and tested on modern web browsers like Google Chrome. However, it should be compatible with other major web browsers such as Mozilla Firefox, Microsoft Edge, and Safari. Testing the system across different browsers ensures cross-browser compatibility.

6. **Package Managers**: Besides NPM, developers may utilize additional package managers like Yarn to manage project dependencies efficiently. These package managers enable easy installation and updating of libraries and tools used in the development process.

7. **Development Server:** A development server is used to serve the web application locally during development. Tools like Vite or webpack-dev-server can be employed to provide hot-reloading capabilities, allowing developers to see real-time changes without manually refreshing the browser.

8. **Debugging and Testing Tools**: Chrome DevTools, as mentioned earlier, is a powerful tool for debugging and testing web applications. It provides features like breakpoints, console logging, network monitoring, and performance profiling. Additionally, automated testing frameworks like Jest or React Testing Library can be used for writing unit tests to ensure code quality and functionality.

By setting up the development environment with the appropriate tools and components, developers can efficiently write, debug, and test the code for the proposed system, ensuring a smooth and productive development experience.

6.3 Software Interface

The software interface of the proposed system comprises the user interface (UI) and the underlying APIs that facilitate communication between different system components.

1. **User Interface (UI):**
   - Grid: The grid is a visual representation of the pathfinding environment. It consists of cells that can be marked as obstacles or open spaces. Users can interact with the grid by selecting cells, creating obstacles, defining the start and target positions, and observing the pathfinding algorithm's progress and results.
   - Algorithm Selection: The UI provides options for users to choose from a set of available pathfinding algorithms, such as Dijkstra's algorithm, Breadth-First Search (BFS), or Depth-First Search (DFS). Users can select the algorithm they want to visualize and analyze.
   - Maze Generation: The system allows users to generate random or recursive mazes within the grid. This feature provides users with pre-defined maze configurations that can serve as interesting pathfinding challenges.
   - Controls: The UI includes controls for starting and stopping the pathfinding process, clearing the grid or path, adjusting the animation speed, and other system-specific functionalities. These controls enable users to interact with the system and modify its behavior as needed.
   - Performance Metrics: The UI displays relevant performance metrics during and after the pathfinding process. This includes information such as the time taken to find the path, the number of cells traveled, and the number of cells scanned by the algorithm. These metrics help users understand the efficiency and effectiveness of different algorithms.
   - Visual Feedback: The UI provides visual feedback to users, such as highlighting the cells being visited, marking the path found by the algorithm, and displaying animations to depict the algorithm's progress. This visual feedback enhances the understanding and visualization of the pathfinding algorithms.

2. APIs:
   - Algorithm API: The system exposes an API that allows the UI to interact with the underlying pathfinding algorithms. This API includes functions to initiate the pathfinding process, retrieve the path or solution found, and obtain algorithm-specific information or metrics.
   - Grid API: The system provides an API to manipulate and query the grid. This includes functions to mark cells as obstacles, define the start and target positions, retrieve the state of cells, and clear the grid.
   - UI Event Handlers: The system implements event handlers to capture user interactions and translate them into actions. These event handlers enable the UI to respond to user input, update the grid and algorithm state, and trigger the visualization of the pathfinding process.

The software interface acts as the bridge between the user and the system, allowing users to interact with the pathfinding visualizer, select algorithms, manipulate the grid, and observe the algorithm's behavior and performance. The underlying APIs enable seamless communication and coordination between the UI components and the core functionality of the syste

6.4 Hardware Used

The proposed system, being a web-based pathfinding visualizer, primarily relies on the hardware components commonly found in modern personal computers and mobile devices. Here are the general hardware requirements for running the system:

1. Computer/Mobile Device: Users can access the pathfinding visualizer using a computer or a mobile device such as a laptop, desktop computer, smartphone, or tablet. The system should be compatible with a wide range of devices across different platforms.

2. Processor: The system can run on processors with varying speeds and capabilities, ranging from basic processors found in entry-level devices to high-performance processors in advanced systems. A processor capable of handling web browsing and running JavaScript-based applications smoothly is sufficient.

3. Memory (RAM): The amount of RAM required depends on the complexity of the grids and the performance of the system. Generally, a minimum of 4 GB of RAM is recommended for smooth execution. However, having more RAM can enhance the system's performance, especially when dealing with larger grids or multiple instances of the pathfinding visualizer.

4. Graphics Processing Unit (GPU): A dedicated GPU is not required for the system to function, as the visual rendering is primarily handled by the web browser. However, having a capable GPU can enhance the overall visual experience, particularly when dealing with animations and graphical effects.

5. Storage: The storage requirements for the system itself are minimal since it is a web-based application that runs directly in the browser. However, a certain amount of storage is necessary to store the browser and the operating system.

6. Internet Connection: A stable internet connection is required to access the system, load the web application, and retrieve any external dependencies or resources. While offline functionality is not a core requirement for the system, the initial loading of the application and resources can be cached for subsequent offline use.

It is important to note that the proposed system does not have any specific hardware dependencies or requirements beyond what is typically found in standard computing devices. The focus is primarily on the software and the performance of the web browser in executing the pathfinding visualizer.

6.5 Algorithms Used in Development

The pathfinding visualizer project incorporates three popular pathfinding algorithms:

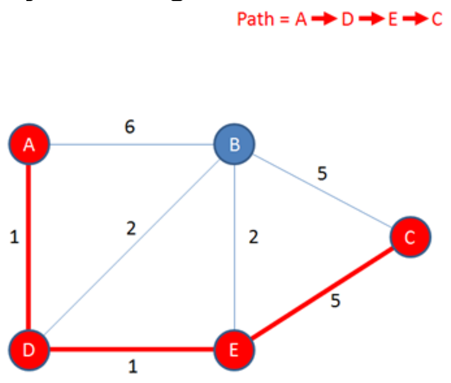
1. Dijkstra's Algorithm:
Dijkstra's algorithm is a widely used algorithm for finding the shortest path between two nodes in a graph. It works by iteratively selecting the node with the lowest cost from a set of unvisited nodes and updating the costs of its neighboring nodes. Dijkstra's algorithm guarantees finding the shortest path in a graph with non-negative edge weights. It is a commonly employed algorithm for solving the single-source shortest path problem.

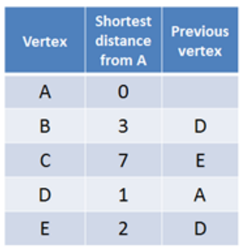
2. Breadth-First Search (BFS):
  Breadth-First Search is an algorithm that explores a graph in a breadthward motion, i.e., it visits all the neighbors of a node before moving to their neighbors. BFS starts at a given source node and visits its neighbors, then moves to their neighbors, and so on. It guarantees finding the shortest path from the source node to all other reachable nodes in an unweighted graph. BFS is commonly used when the graph is unweighted or all edge weights are equal.

3. Depth-First Search (DFS):
  Depth-First Search is an algorithm that explores a graph by traversing as far as possible along each branch before backtracking. It starts at a given source node and explores as deeply as possible along each branch before returning. DFS is often used for graph traversal and can be adapted for various applications, including pathfinding. While DFS does not guarantee finding the shortest path, it is useful for exploring all possible paths in a graph.

These three algorithms offer different approaches to pathfinding, each with its own strengths and characteristics. By including Dijkstra's algorithm, BFS, and DFS in the pathfinding visualizer, users can compare and contrast the behaviors and performances of these algorithms in different scenarios.

Dijkstra's Algorithm:





Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

Dijkstra's algorithm is a popular and efficient algorithm for finding the shortest path between two nodes in a graph. It works on graphs with non-negative edge weights and guarantees finding the shortest path from a source node to all other nodes in the graph. Here's a step-by-step explanation of how Dijkstra's algorithm works:

1. Initialize:
  - Create a set of unvisited nodes and set the distance of the source node to 0. Set the distance of all other nodes to infinity.
  - Set the source node as the current node.

2. Update Distances:
  - For the current node, examine all its neighbors (adjacent nodes) that are still unvisited.
  - For each neighbor, calculate the tentative distance by adding the current node's distance to the edge weight between the current node and the neighbor.
  - If the tentative distance is less than the neighbor's current distance, update the neighbor's distance to the tentative distance.

3. Visit the Next Node:

- Once all the neighbors of the current node have been examined, mark the current node as visited.
- Among the unvisited nodes, select the one with the smallest distance as the next current node.
- Repeat steps 2 and 3 until all nodes have been visited or the target node is reached.

4. Path Reconstruction:
   - Once the target node is reached or all nodes have been visited, the algorithm terminates.
   - To obtain the shortest path from the source to the target node, start at the target node and work backward, following the nodes with the smallest distances until the source node is reached.
   - The resulting path will be the shortest path from the source to the target node, and the total distance will be the sum of the distances of the nodes along the path.

Dijkstra's algorithm provides an optimal solution for finding the shortest path in a graph with non-negative edge weights. Its efficiency depends on the data structure used to store and retrieve the nodes and their distances. Priority queues or min-heaps are commonly used to optimize the algorithm's performance by efficiently selecting the node with the smallest distance in each iteration.

By implementing Dijkstra's algorithm in the pathfinding visualizer, users can observe the step-by-step exploration of the graph and visualize the shortest path from a selected source node to any other node in the graph.

Advantages of Dijkstra's Algorithm:
1. Guaranteed Shortest Path: Dijkstra's algorithm guarantees to find the shortest path between a source node and all other nodes in a graph with non-negative edge weights. It provides an optimal solution in terms of minimizing the total weight or distance of the path.

2. Versatility: Dijkstra's algorithm can be applied to a wide range of graph types, including directed and undirected graphs. It handles both weighted and unweighted graphs, although it performs best with non-negative weights.

3. Wide Usage: Dijkstra's algorithm is widely used in various applications, such as network routing, transportation planning, GPS navigation systems, and graph analysis. Its reliability and efficiency make it a popular choice in scenarios where finding the shortest path is critical.

4. Efficiency with Min Heap/Priority Queue: By using a min heap or a priority queue to optimize the selection of the next node, Dijkstra's algorithm achieves a time complexity of $O((V + E) \log V)$, where V is the number of vertices and E is the number of edges. This efficiency makes it suitable for large graphs.

Disadvantages of Dijkstra's Algorithm:
1. Non-Negative Edge Weights: Dijkstra's algorithm requires non-negative edge weights to ensure the correctness of the shortest path. If the graph contains negative edge weights, the algorithm may produce incorrect results or even enter an infinite loop.

2. Inefficient with Negative Edge Weights: In graphs with negative edge weights, Dijkstra's algorithm fails to find the correct shortest path. For such scenarios, alternative algorithms like the Bellman-Ford algorithm or the A* algorithm with appropriate heuristics should be used.

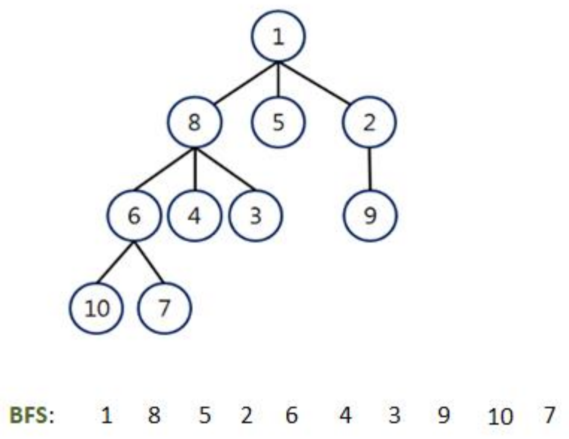
3. Space Complexity: Dijkstra's algorithm requires additional space to store the distances from the source node to all other nodes. In large graphs, this can lead to increased memory consumption.

4. Lack of Parallelism: Dijkstra's algorithm does not naturally lend itself to parallelization. Each node's distance calculation depends on the distances of its neighbors, making parallel execution more challenging.

5. No Handling of Cycles: Dijkstra's algorithm assumes that there are no negative cycles in the graph. If a negative cycle exists, the algorithm may end up in an infinite loop or produce incorrect results.

It is important to consider these advantages and disadvantages when deciding whether to use Dijkstra's algorithm for a particular graph traversal or shortest path problem. The specific characteristics of the graph, such as edge weights and presence of negative cycles, should be carefully analyzed to determine the suitability of the algorithm.

Breadth-First Search (BFS):



Breadth-First Search (BFS) is an algorithm used to traverse or search through a graph in a breadthward motion. It explores all the neighbors of a node before moving on to their neighbors, thus covering the graph layer by layer. BFS is often employed for finding the shortest path in an unweighted graph or exploring all reachable nodes from a given source node. Here's how BFS works:

1. Initialize:
   - Create a queue to store nodes for traversal.
   - Enqueue the source node into the queue.
   - Create a set to track visited nodes and mark the source node as visited.

2. Traverse:
   - Dequeue a node from the front of the queue.
   - Process the node (perform desired operations).
   - Enqueue all unvisited neighbors of the node into the queue and mark them as visited.
   - Repeat this process until the queue is empty.

3. Termination:
   - The algorithm terminates when the queue becomes empty, indicating that all reachable nodes have been traversed.

BFS guarantees finding the shortest path from the source node to all other reachable nodes in an unweighted graph. It explores nodes layer by layer, ensuring that nodes closer to the source are visited before nodes further away. By storing the parent of each visited node during traversal, the shortest path from the source node to any other node can be easily reconstructed.

In the context of the pathfinding visualizer, BFS can be used to visualize the breadth-first exploration of the grid and observe the shortest path from a selected source node to any other reachable node. It is particularly useful when the graph is unweighted or all edge weights are considered equal. BFS provides a systematic and comprehensive way to explore the graph, making it an important algorithm in the domain of pathfinding and graph traversal.

Advantages of Breadth-First Search (BFS):

1. Guaranteed Shortest Path for Unweighted Graphs: BFS guarantees to find the shortest path between a source node and any other reachable node in an unweighted graph. It explores vertices in breadth-first order, ensuring that the shortest path is discovered as it traverses the graph layer by layer.

2. Completeness: BFS will visit every reachable vertex in the graph, ensuring that no vertex is left unexplored if it is connected to the source vertex. This makes BFS a complete algorithm for exploring all reachable nodes.

3. Minimal Memory Usage: BFS only requires memory to store the visited set and the queue for tracking vertices to visit. It does not require additional data structures such as priority queues or heaps, making it memory-efficient for large graphs.

4. Applications in Other Algorithms: BFS serves as a building block for various graph algorithms. It is used in algorithms like Dijkstra's algorithm, A* search, and network flow algorithms to solve more complex problems.

5. Identifying Connected Components: BFS can be used to identify connected components in a graph. By performing multiple BFS traversals from different starting points, it can determine all the connected components in the graph.

Disadvantages of Breadth-First Search (BFS):

1. Memory Usage for Dense Graphs: In dense graphs, where the number of edges is close to the maximum possible number of edges, BFS may consume significant memory. This is because the queue needs to store a large number of vertices.

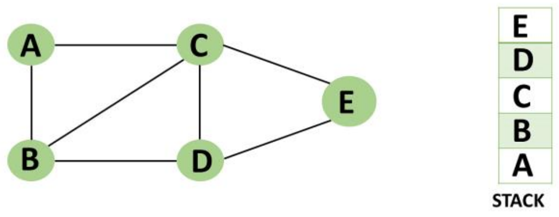
2. Not Suitable for Pathfinding in Weighted Graphs: BFS is not designed to find the shortest path in graphs with weighted edges. While it can be adapted for weighted graphs by using techniques like uniform edge weights or modifying the algorithm, other algorithms like Dijkstra's algorithm or A* search are more suitable for weighted pathfinding problems.

3. Lack of Precision in Certain Scenarios: BFS treats all edges equally and explores them in the order they are encountered. In some cases, this may lead to suboptimal results or an inefficient search path. Algorithms like Dijkstra's algorithm, which take edge weights into account, provide more precise and efficient pathfinding in weighted graphs.

4. Exponential Time Complexity in Branching Factor: BFS has a time complexity of O(V + E), where V is the number of vertices and E is the number of edges. In graphs with high branching factors, the number of vertices to explore can grow exponentially, leading to longer execution times.

It's important to consider these advantages and disadvantages when deciding whether to use BFS for a specific graph traversal or pathfinding problem. The characteristics of the graph, such as edge weights, density, and desired output, should be carefully evaluated to determine the most appropriate algorithm to use.

Depth-First Search (DFS):



Depth-First Search (DFS) is an algorithm used to traverse or search through a graph by exploring as far as possible along each branch before backtracking. It starts at a given source node and explores as deeply as possible before returning and exploring other branches. DFS is commonly used for graph traversal and can be adapted for various applications, including pathfinding. Here's how DFS works:

1. Initialize:
   - Create a stack to store nodes for traversal.
   - Push the source node onto the stack.
   - Create a set to track visited nodes and mark the source node as visited.

2. Traverse:
   - Pop a node from the top of the stack.
   - Process the node (perform desired operations).
   - Push all unvisited neighbors of the node onto the stack and mark them as visited.
   - Repeat this process until the stack is empty.

3. Termination:
   - The algorithm terminates when the stack becomes empty, indicating that all reachable nodes have been traversed.

DFS does not guarantee finding the shortest path like Dijkstra's algorithm or BFS. However, it is useful for exploring all possible paths in a graph and can be adapted to find specific paths based on certain conditions. DFS may get trapped in infinite loops if the graph contains cycles, so it is essential to mark nodes as visited to avoid revisiting them.

In the pathfinding visualizer, DFS can be employed to explore the grid and visualize the depth-first traversal of the graph. It allows users to observe the exploration of different paths and provides insights into the connectivity of the graph. While not designed for finding the shortest path, DFS serves as a valuable tool for analyzing graph structures and can be used in combination with other algorithms to achieve specific pathfinding goals.

Advantages of Depth-First Search (DFS):

1. Memory Efficiency: DFS typically uses less memory compared to breadth-first search (BFS) as it only needs to store information about the current path and backtrack when necessary. This makes it more memory-efficient for large graphs or recursive implementations.

2. Simplicity of Implementation: DFS has a straightforward recursive implementation, making it easy to understand and implement. It requires minimal bookkeeping and data structures, simplifying the coding process.

3. Discovery of Multiple Solutions: DFS can be useful in scenarios where multiple solutions or paths need to be explored. It can exhaustively search through all possible paths, allowing for the discovery of various solutions or satisfying certain conditions.

4. Identifying Cycles and Backtracking: DFS is well-suited for identifying cycles in a graph, as it backtracks whenever it encounters a visited node. This property makes it valuable in applications such as cycle detection or topological sorting.

Disadvantages of Depth-First Search (DFS):

1. Completeness is not Guaranteed: Unlike BFS, DFS does not guarantee that it will find a solution if one exists. Depending on the structure of the graph and the chosen traversal path, DFS might get stuck in an infinite loop or fail to explore all reachable nodes.

2. Suboptimal Solutions: DFS does not always find the shortest path or the most optimal solution. It tends to prioritize exploring deep paths before considering shallower paths. In cases where the shortest path or the optimal solution is required, algorithms like Dijkstra's algorithm or A* search are more suitable.

3. Lack of Uniformity in Path Lengths: DFS can lead to variable path lengths depending on the traversal path chosen. This can result in suboptimal or inconsistent results, especially in graphs with varying edge weights.

4. Stack Overflow in Recursive Implementations: Recursive implementations of DFS may encounter stack overflow errors when exploring deep paths in very large or deeply nested

graphs. This can limit its practicality for certain scenarios or require additional measures like tail recursion optimization.

5. No Breadth-First Characteristics: DFS does not exhibit the breadth-first nature of exploring neighboring vertices before moving deeper into the graph. This can lead to an unbalanced exploration of the graph and possibly missing important nodes or paths.

When deciding to use DFS, it is crucial to consider the specific characteristics of the graph and the requirements of the problem at hand. DFS is often favored for its simplicity and memory efficiency, but its limitations, such as incompleteness and suboptimal solutions, need to be taken into account.

7. DESIGNS

To provide a comprehensive understanding of the proposed pathfinding visualizer, the system design encompasses several key diagrams that illustrate the data flow, user interactions, and database structure. These diagrams include the Data Flow Diagram (DFD), Use Case Diagram, and Entity-Relationship (ER) Diagram.

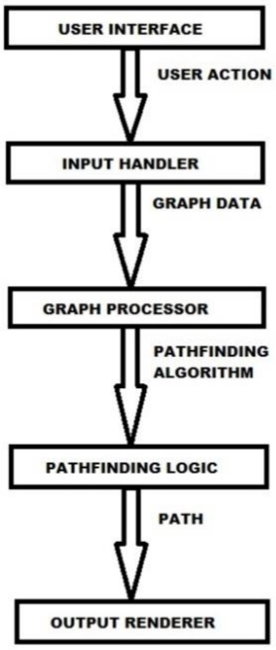
1. Data Flow Diagram (DFD):
The DFD illustrates the flow of data within the system, showcasing the processes, data stores, and data flows involved. In the context of the pathfinding visualizer, the DFD highlights the interactions between the user interface, algorithms, and the grid. It demonstrates how data is passed between these components during the pathfinding process.

2. Use Case Diagram:
The Use Case Diagram outlines the various interactions and relationships between the system and its users. It identifies the different types of users and the actions they can perform within the system. In the pathfinding visualizer, the Use Case Diagram showcases user actions such as selecting algorithms, creating obstacles, generating mazes, and controlling the visualization process.

3. Entity-Relationship (ER) Diagram:
The ER Diagram illustrates the relationships between entities in the system and their attributes. Although the pathfinding visualizer may not require an extensive database, an ER Diagram can still represent the entities and their relationships. For instance, it can capture the relationships between users, their preferences, and any saved configurations or paths.

These design artifacts provide a comprehensive view of the system's architecture, functionality, and data flow. They aid in understanding how users interact with the system, how data is processed and transferred, and how the system is structured from a database perspective. By utilizing these design diagrams, stakeholders and developers can align their understanding of the system and facilitate effective communication during the development process.

7.1 Data Flow Diagram



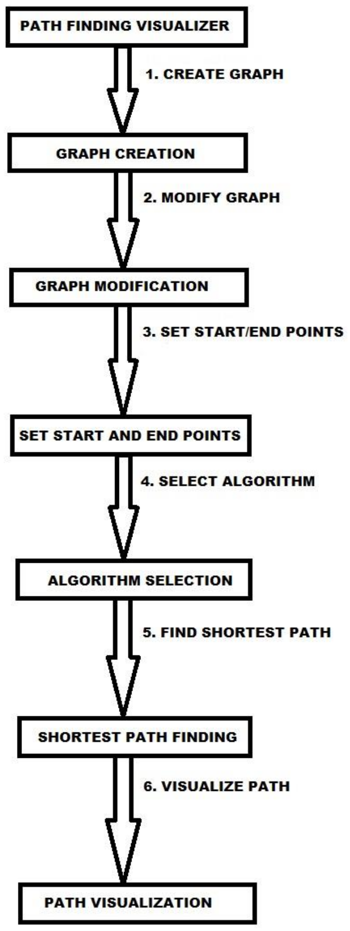
Explanation of components:

1. User Interface: This component represents the visual interface through which the user interacts with the path finding visualizer. It includes elements like buttons, input fields, and the graphical representation of the graph.
2. Input Handler: The input handler receives user actions from the user interface and converts them into graph data. It captures user input for creating or modifying the graph, such as adding nodes, defining connections, setting start and end points, and selecting algorithms.
3. Graph Processor: The graph processor component takes the graph data from the input handler and performs necessary processing tasks. This may include validating the graph, optimizing data structures for efficient pathfinding, and managing the overall graph representation.
4. Pathfinding Logic: This component houses the core pathfinding algorithms such as Dijkstra's algorithm, BFS, DFS, or other algorithms. It receives the processed graph data from the graph processor and applies the selected algorithm to find the shortest path between the start and end points.
5. Output Renderer: The output renderer component takes the resulting path from the pathfinding logic and renders it visually on the user interface. It may highlight the shortest path, visualize the traversal process, and provide other relevant information to the user.

The data flow starts from the user interface, where the user interacts with the visualizer, and flows through the input handler, graph processor, pathfinding logic, and output renderer. Each component performs its specific tasks and passes the data to the next component in the flow. Finally, the resulting path is displayed to the user through the output renderer.

Please note that this is a simplified representation of the data flow in a path finding visualizer. The actual implementation may involve additional components, data structures, and interactions depending on the specific requirements and design choices.

7.2 Use-Case Diagram



Explanation of use cases:

1. Create Graph: The user can create a new graph by defining nodes and their connections. This use case allows the user to set up the initial graph structure.
2. Modify Graph: The user can modify the existing graph by adding, deleting, or modifying nodes and connections. This use case enables the user to refine the graph structure.
3. Set Start/End Points: The user can specify the starting and ending points on the graph. This use case allows the user to set the points between which the shortest path needs to be found.
4. Select Algorithm: The user can choose a specific pathfinding algorithm (e.g., Dijkstra's algorithm, BFS, DFS) from the available options. This use case enables the user to select the algorithm for finding the shortest path.

5. Find Shortest Path: The system applies the selected algorithm to find the shortest path between the specified start and end points on the graph. This use case represents the core functionality of the pathfinding process.
6. Visualize Path: Once the shortest path is found, the system visualizes it on the graph, highlighting the nodes and connections that make up the path. This use case provides a visual representation of the shortest path to the user.

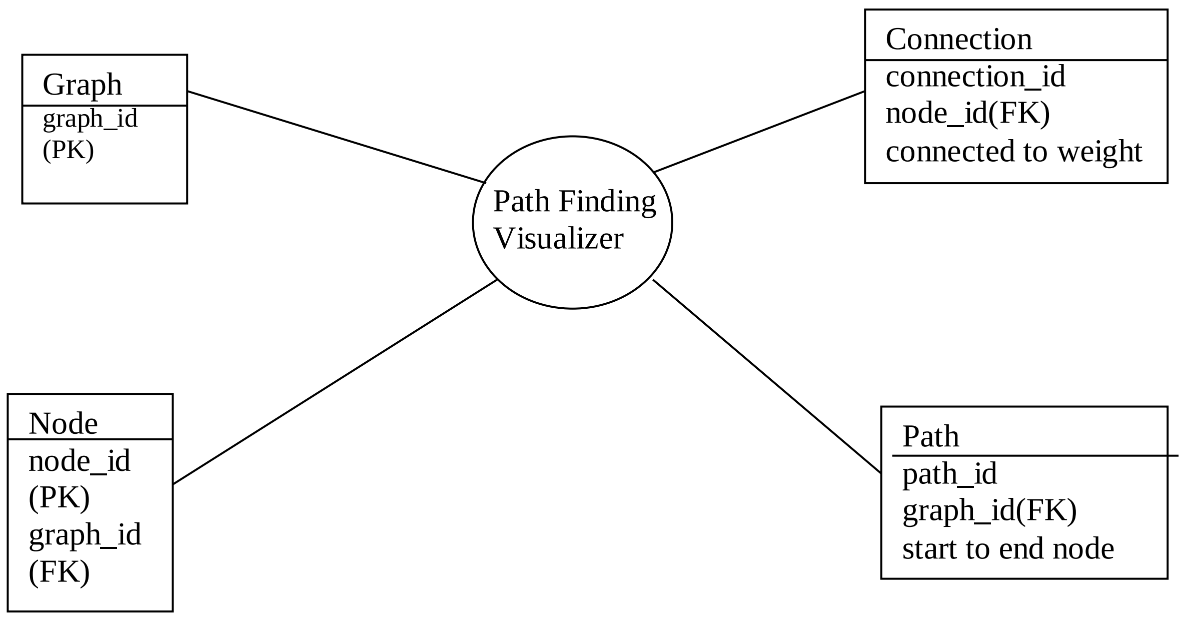
These use cases capture the main functionalities of the path finding visualizer, allowing the user to create and modify graphs, set start and end points, select algorithms, find the shortest path, and visualize the results.

7.3 Entity Relationship Diagram

Explanation of entities:

1. Graph: Represents a graph instance created by the user. It has a primary key graph_id to uniquely identify each graph.
2. Node: Represents a node in the graph. Each node has a primary key node_id and is associated with a specific graph through the foreign key graph_id.
3. Connection: Represents a connection (edge) between nodes in the graph. It has a primary key connection_id and is associated with a node through the foreign key node_id. The connected_to field specifies the node to which the connection is made, and the weight field represents the weight or cost of the connection.
4. Path: Represents a specific path in the graph. It has a primary key path_id and is associated with a graph through the foreign key graph_id. The start_node and end_node fields define the starting and ending nodes of the path, respectively.

This simplified ER diagram captures the basic entities and their relationships in a path finding visualizer. Additional entities or attributes can be added based on specific requirements, such as user accounts, saved graphs, or algorithm preferences.



8 DATA DICTIONARY

1. User:
   - Description: Represents a user of the pathfinding visualizer system.
   - Attributes:
   - UserID: A unique identifier for each user. (Primary Key)

2. Algorithm:
   - Description: Represents a pathfinding algorithm available in the system.
   - Attributes:
   - AlgorithmID: A unique identifier for each algorithm. (Primary Key)
   - AlgorithmName: The name or title of the algorithm.
   - Description: A brief description or summary of the algorithm.

3. Grid:
   - Description: Represents a grid used for pathfinding visualization.
   - Attributes:
   - GridID: A unique identifier for each grid. (Primary Key)
   - GridName: The name or title of the grid.
   - Dimensions: The dimensions of the grid (e.g., rows and columns).
   - UserID: The ID of the user who created or owns the grid. (Foreign Key referencing User.UserID)

4. Path:
   - Description: Represents a path or solution found by a pathfinding algorithm.
   - Attributes:
   - PathID: A unique identifier for each path. (Primary Key)
   - GridID: The ID of the grid associated with the path. (Foreign Key referencing Grid.GridID)
   - UserID: The ID of the user who initiated the pathfinding process. (Foreign Key referencing User.UserID)
   - TimeTaken: The time taken by the algorithm to find the path.
   - CellsTraveled: The number of cells traversed to reach the target.
   - CellsScanned: The number of cells evaluated by the algorithm.

9. TESTING

Testing is an essential part of the software development process for ensuring the quality, functionality, and reliability of the pathfinding visualizer system. It involves validating the system's behavior, identifying and fixing defects, and ensuring that it meets the specified requirements. Here are some types of testing that can be performed for the pathfinding visualizer:

1. Unit Testing:
   - Test individual units or components of the system, such as algorithms, data structures, or helper functions.
   - Verify that each unit behaves as expected and produces the correct output for a given input.
   - Use testing frameworks and tools specific to the programming language and technology stack being used.

2. Integration Testing:
   - Test the integration of different components of the system, such as the user interface, algorithms, and data structures.
   - Validate that the components work together as expected and exchange data correctly.
   - Ensure that the system behaves consistently and correctly across different components.

3. Functional Testing:
   - Test the system's functionality against the defined requirements and use cases.
   - Verify that the pathfinding algorithms correctly find the optimal path from the start position to the target position.
   - Validate that users can interact with the system, select algorithms, create obstacles, generate mazes, and observe the visualization.

4. Performance Testing:
   - Evaluate the system's performance and responsiveness under different scenarios and loads.
   - Measure and analyze the system's behavior in terms of execution time, memory usage, and responsiveness.
   - Identify and optimize any bottlenecks or performance issues to ensure smooth operation.

5. Usability Testing:
   - Evaluate the user interface and user experience of the pathfinding visualizer.
   - Gather feedback from users to assess the system's ease of use, intuitiveness, and effectiveness.
   - Identify any usability issues or areas for improvement and make necessary adjustments.

6. Regression Testing:
   - Re-run previously executed tests to ensure that the modifications or additions to the system did not introduce new defects or regressions.

- Validate that the existing functionalities continue to work as expected after changes have been made.

7. Compatibility Testing:
   - Test the compatibility of the pathfinding visualizer with different web browsers, operating systems, and devices.
   - Ensure that the system functions correctly and displays properly across various platforms and configurations.

8. Security Testing:
   - Evaluate the system's security measures to identify potential vulnerabilities or weaknesses.
   - Perform penetration testing, code reviews, and other security assessments to ensure data privacy and protection.

Throughout the testing process, it is important to document test cases, track defects, and maintain a systematic approach to testing. By conducting thorough and comprehensive testing, you can ensure the reliability and robustness of the pathfinding visualizer system.

9.1 Unit Testing

Unit testing for a web application involves testing the individual units or components of the application, such as functions, modules, or classes, to verify their correctness and identify any bugs or errors. In the context of the pathfinding visualizer web app, unit testing can be performed to verify the functionality and accuracy of the three algorithms: Dijkstra's algorithm, Breadth-First Search (BFS), and Depth-First Search (DFS). Here's an approach to unit testing for algorithm verification and bug detection:

1. Test Case Design:
- Design test cases to cover different scenarios and edge cases for each algorithm.
- Define test cases that include various grid configurations, start and target positions, and obstacle placements.
- Consider both expected path outcomes and potential failure cases to ensure comprehensive testing.

2. Test Setup:
- Set up a controlled test environment specifically for unit testing.
- Create test grids with known configurations and expected outcomes for each algorithm.
- Ensure that the test environment isolates the algorithm under test from other components.

3. Execute Test Cases:
- Implement the test cases using a unit testing framework suitable for web applications, such as Jest or Mocha.
- Provide the necessary inputs, such as grid configurations and algorithm selection, for each test case.
- Execute the algorithms on the test grids and compare the actual paths with the expected paths.

4. Assertions and Error Handling:

- Use assertions or built-in assertion libraries to verify the correctness of the algorithm's output.
- Compare the actual paths obtained from the algorithm with the expected paths for each test case.
- Detect and handle any errors or exceptions that occur during the test execution.

5. Bug Detection and Error Reporting:
- Monitor the test results for any failed test cases or unexpected behavior.
- Debug and investigate the failed test cases to identify the source of the bug or error.
- Report any identified bugs or errors in a systematic manner, including detailed descriptions and steps to reproduce.

6. Test Coverage and Continuous Integration:
- Aim for high test coverage to ensure comprehensive testing of the algorithms.
- Continuously update and expand the unit test suite to cover additional scenarios or algorithm enhancements.
- Integrate the unit tests into a continuous integration (CI) pipeline to automate the testing process and receive feedback on code changes.

By performing unit testing on the web application and specifically targeting the algorithm implementation, you can verify the correctness of the three algorithms (Dijkstra's, BFS, and DFS) and detect any bugs or errors that may affect the pathfinding functionality. Unit testing helps ensure the reliability and accuracy of the pathfinding visualizer web app by validating the behavior of individual components in isolation.

9.2 Integrity Testing

Integrity testing is a type of testing that focuses on verifying the integrity and stability of a system by subjecting it to a high load and ensuring that it can handle multiple user interactions without errors or issues. In the context of the pathfinding visualizer, integrity testing can be performed to validate the robustness of the system and the correctness of the algorithms by simulating many users and testing various scenarios. Here's an approach to conducting integrity testing:

1. Test Scenario Design:
- Define various test scenarios that simulate a high load on the system.
- Design test cases that involve multiple users interacting with the pathfinding visualizer simultaneously.
- Include scenarios such as concurrent pathfinding requests, concurrent obstacle creations, and random user interactions.

2. Test Environment Setup:
- Set up a testing environment that closely resembles the production environment.
- Ensure that the system is configured to handle a high number of concurrent users and requests.
- Deploy the system on a server or hosting platform that can support the expected load.

3. Load Generation:
- Use load testing tools or frameworks.

- Configure the load testing tool to simulate multiple users simultaneously performing actions on the system.
- Define the desired load parameters, such as the number of concurrent users, request rates, and think times.

4. Test Execution:
- Execute the integrity tests using the defined test scenarios and load parameters.
- Monitor the system's behavior during the test execution for any errors, crashes, or unexpected behavior.
- Observe the performance metrics, such as response times, CPU and memory usage, and error rates.

5. Error Detection and Handling:
- Detect and analyze any errors or issues that occur during the test execution.
- Identify the root causes of errors, whether they are related to the algorithms, data handling, or system scalability.
- Implement appropriate error handling mechanisms to gracefully handle errors and prevent system crashes.

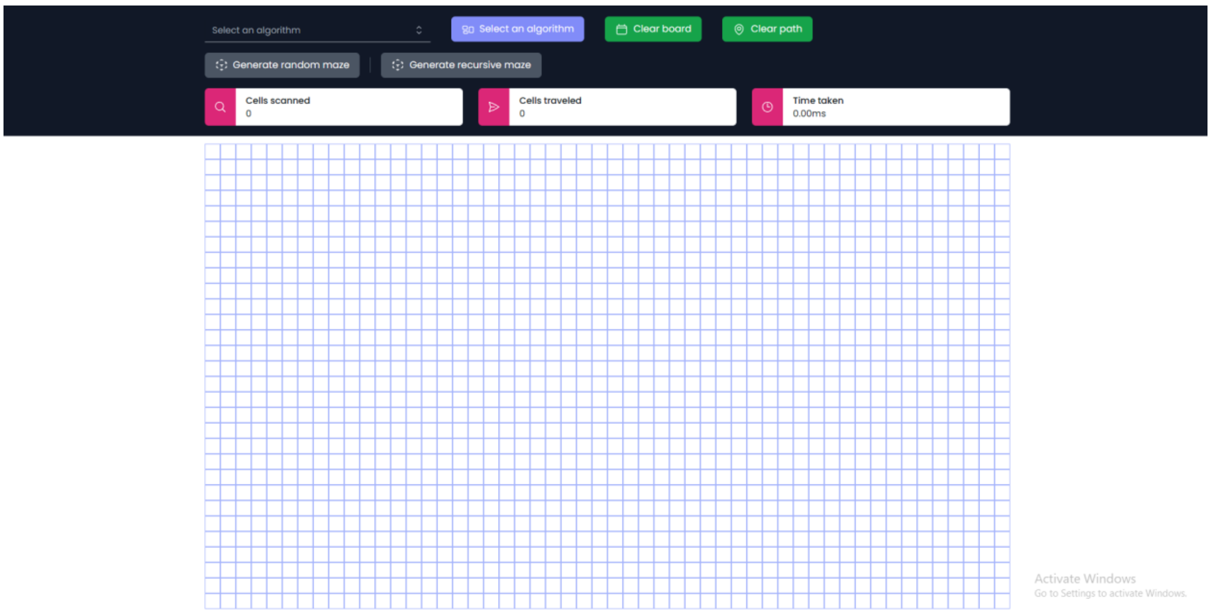
6. Scalability and Performance Analysis:
- Assess the system's scalability by gradually increasing the load and observing how it handles the growing user base.
- Analyze the system's performance under high load conditions and identify any performance bottlenecks or resource limitations.
- Optimize the system's performance by addressing any identified scalability or performance issues.

7. Test Reporting:
- Document the test results, including any errors, performance metrics, and observations.
- Report any identified issues or limitations in a systematic manner, including steps to reproduce and potential resolutions.
- Provide recommendations for improvements and optimizations to enhance the system's integrity and performance.

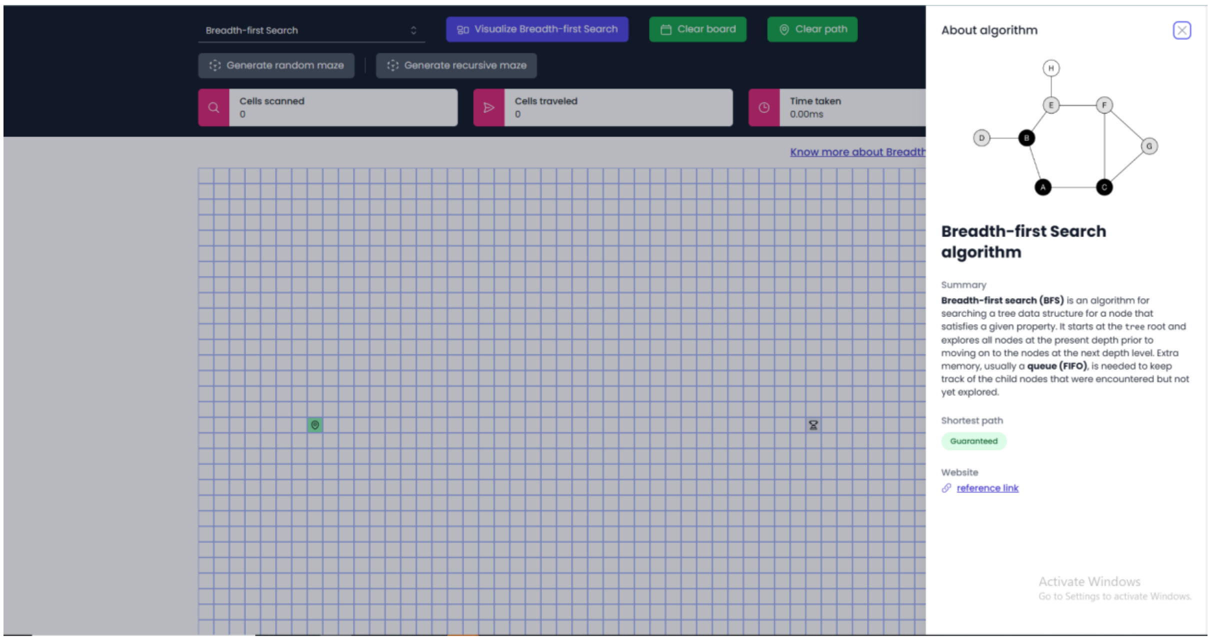
Integrity testing helps validate the stability and correctness of the pathfinding visualizer system under a heavy load, ensuring that it can handle multiple users and interactions without errors or unexpected behavior. By subjecting the system to rigorous testing, you can identify any weaknesses or limitations, and take appropriate measures to improve its integrity and performance.

10. SNAPSHOTS

Home Page



Know more about BFS



Breadth-first Search algorithm

Summary

Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the `tree` root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a **queue (FIFO)**, is needed to keep track of the child nodes that were encountered but not yet explored.

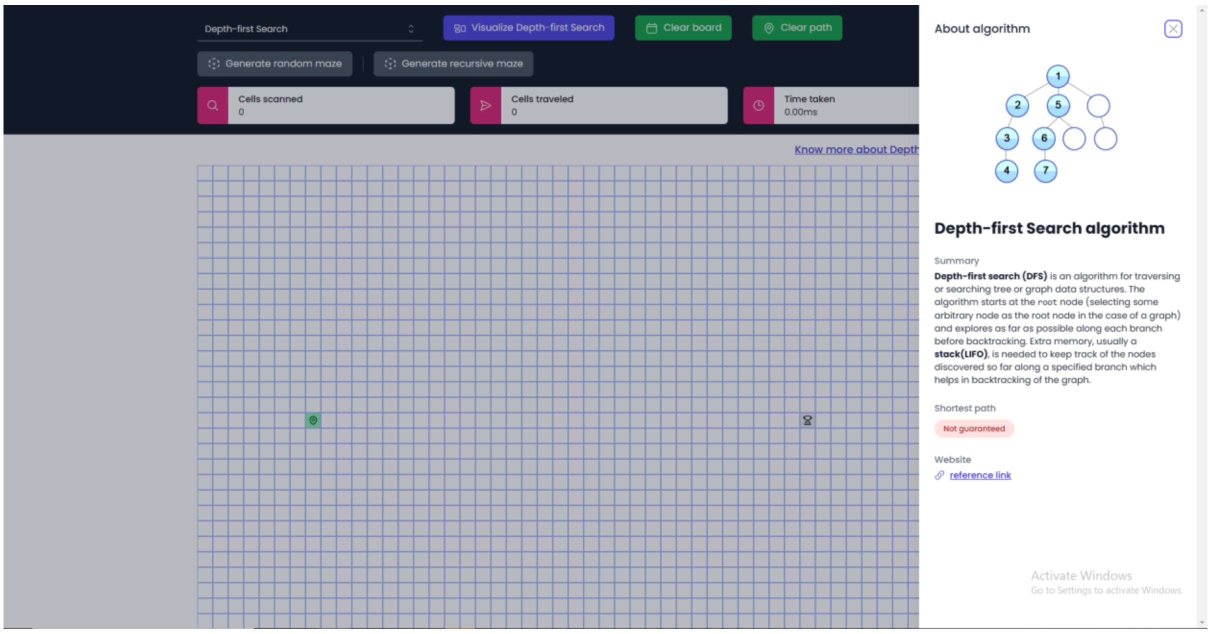
Shortest path

Guaranteed

Website

reference link

Know more about DFS



About algorithm

Depth-first Search algorithm

Summary

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. Extra memory, usually a **stack(LIFO)**, is needed to keep track of the nodes discovered so far along a specified branch which helps in backtracking of the graph.

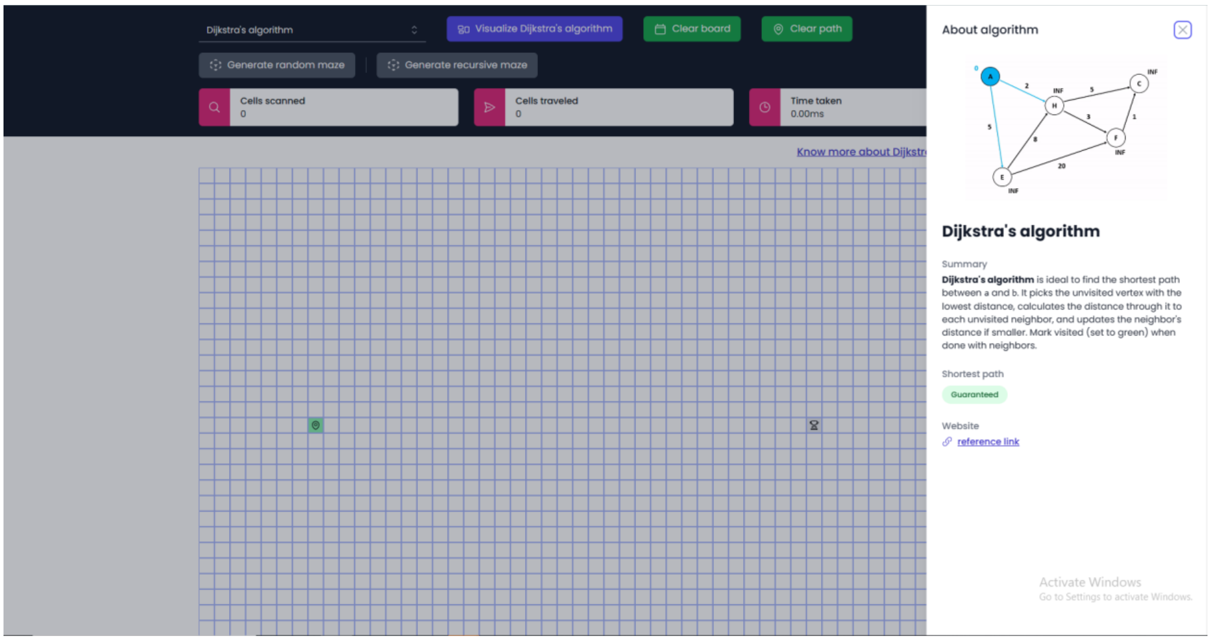
Shortest path

Not guaranteed

Website

reference link

Know more about Dijkastra



About algorithm

Dijkstra's algorithm

Summary

Dijkstra's algorithm is ideal to find the shortest path between a and b. It picks the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited (set to green) when done with neighbors.

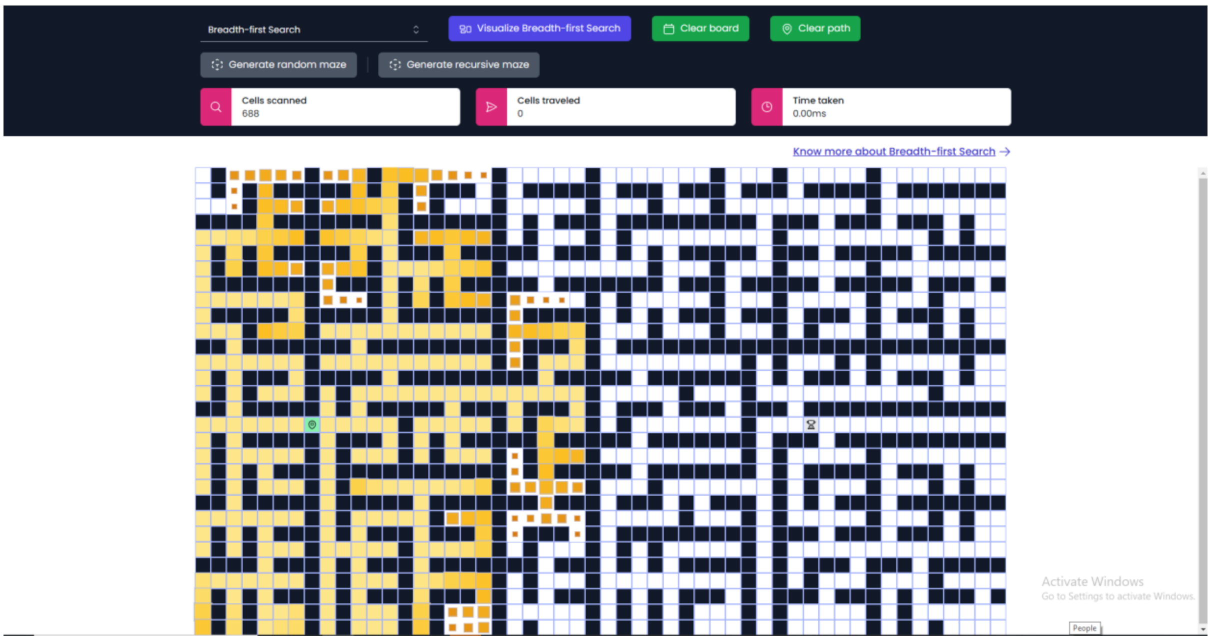
Shortest path

Guaranteed

Website

reference link

Path Finding by BFS Algorithm


Breadth-first Search

Visualize Breadth-first Search | Clear board | Clear path

Generate random maze | Generate recursive maze

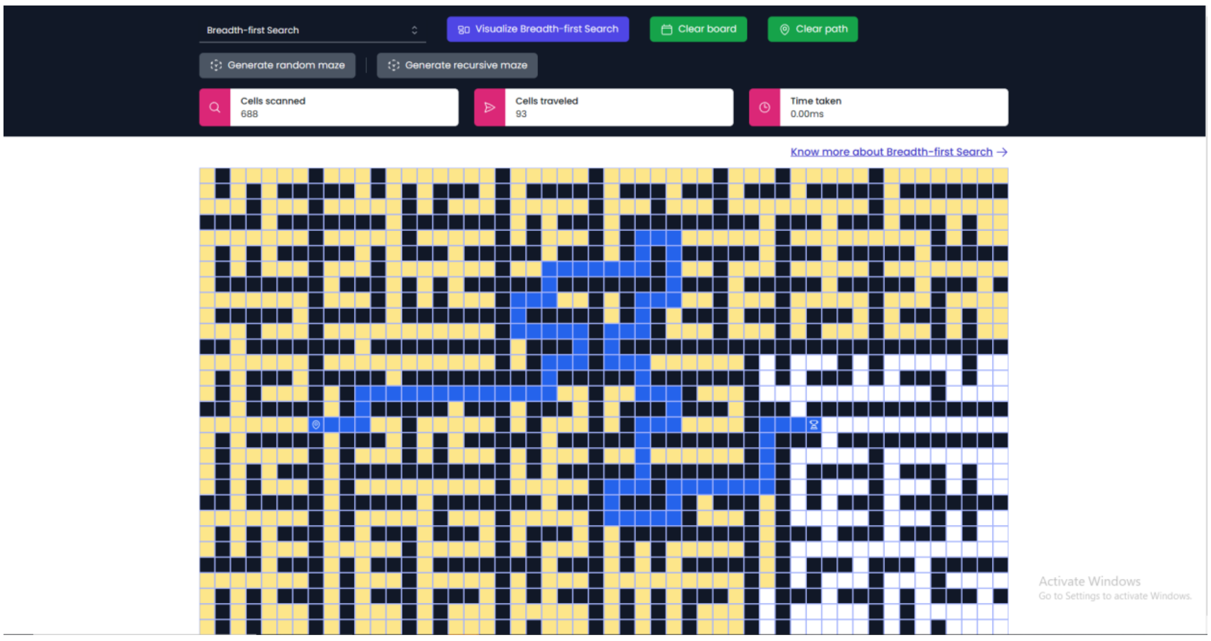
Cells scanned
688

Cells traveled
0

Time taken
0.00ms

Know more about Breadth-first Search →

Path founded by BFS Algorithm


Breadth-first Search

Visualize Breadth-first Search | Clear board | Clear path

Generate random maze | Generate recursive maze

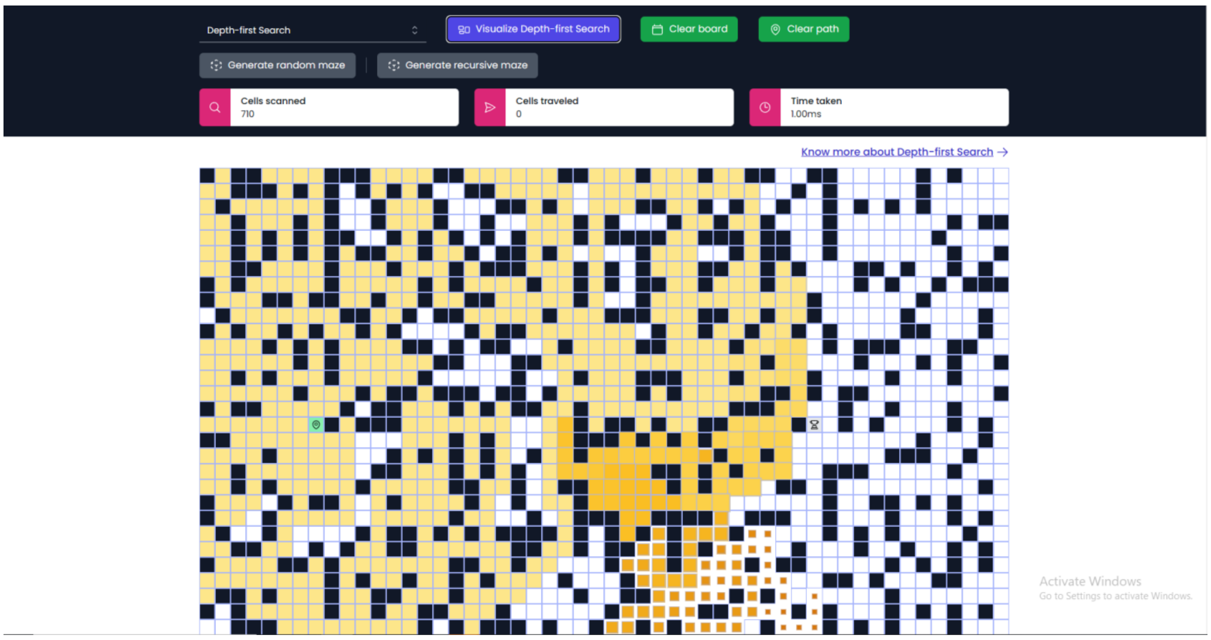
Cells scanned
688

Cells traveled
93

Time taken
0.00ms

Know more about Breadth-first Search →

Path Finding by DFS Algorithm

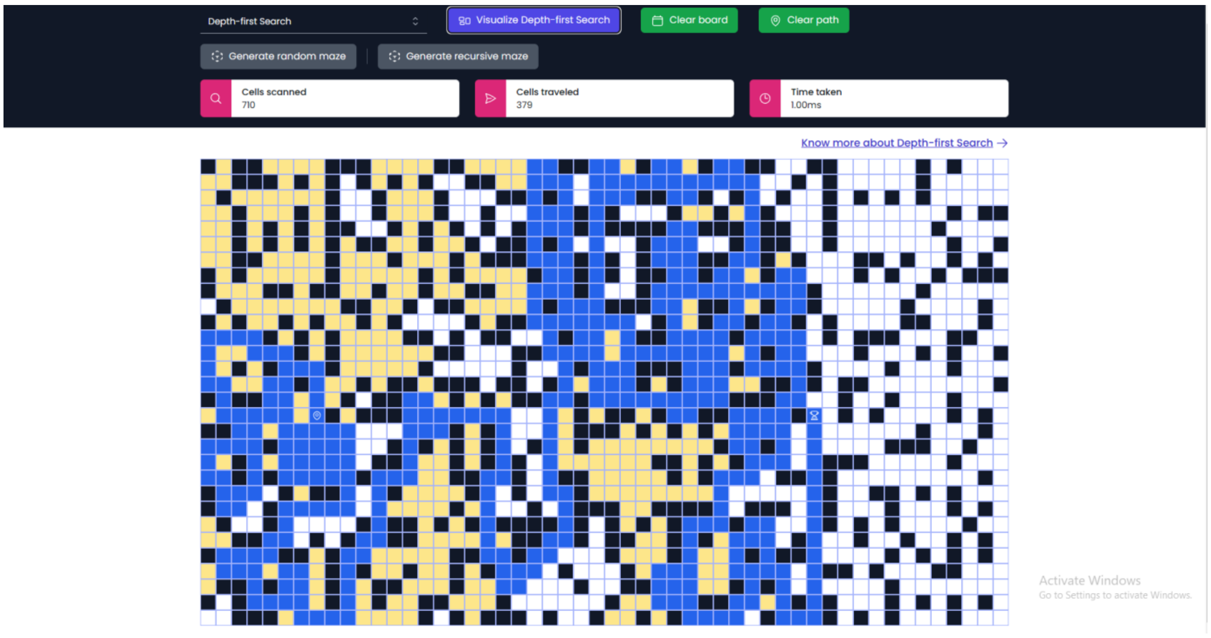

Depth-first Search
Visualize Depth-first Search
Clear board
Clear path
Generate random maze
Generate recursive maze
Know more about Depth-first Search →
Activate Windows
Go to Settings to activate Windows.

Cells scanned 710

Cells traveled 0

Time taken 1.00ms

Path founded by DFS Algorithm

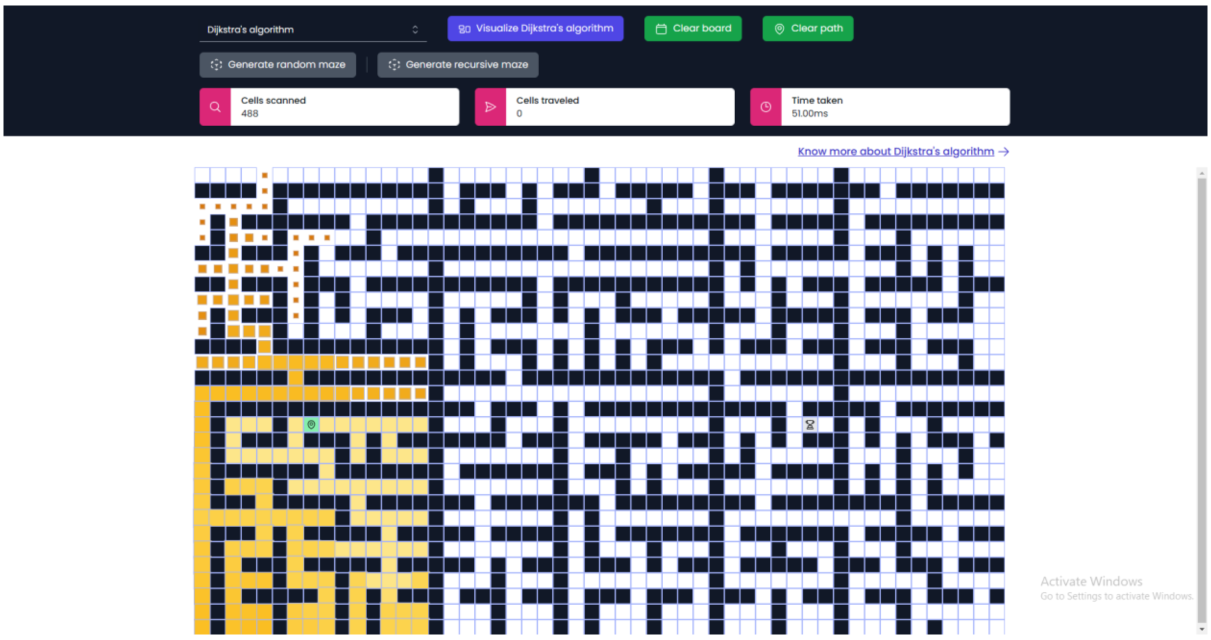

Depth-first Search
Visualize Depth-first Search
Clear board
Clear path
Generate random maze
Generate recursive maze
Know more about Depth-first Search →
Activate Windows
Go to Settings to activate Windows.

Cells scanned 710

Cells traveled 379

Time taken 1.00ms

Path Finding by Dijkastra Algorithm



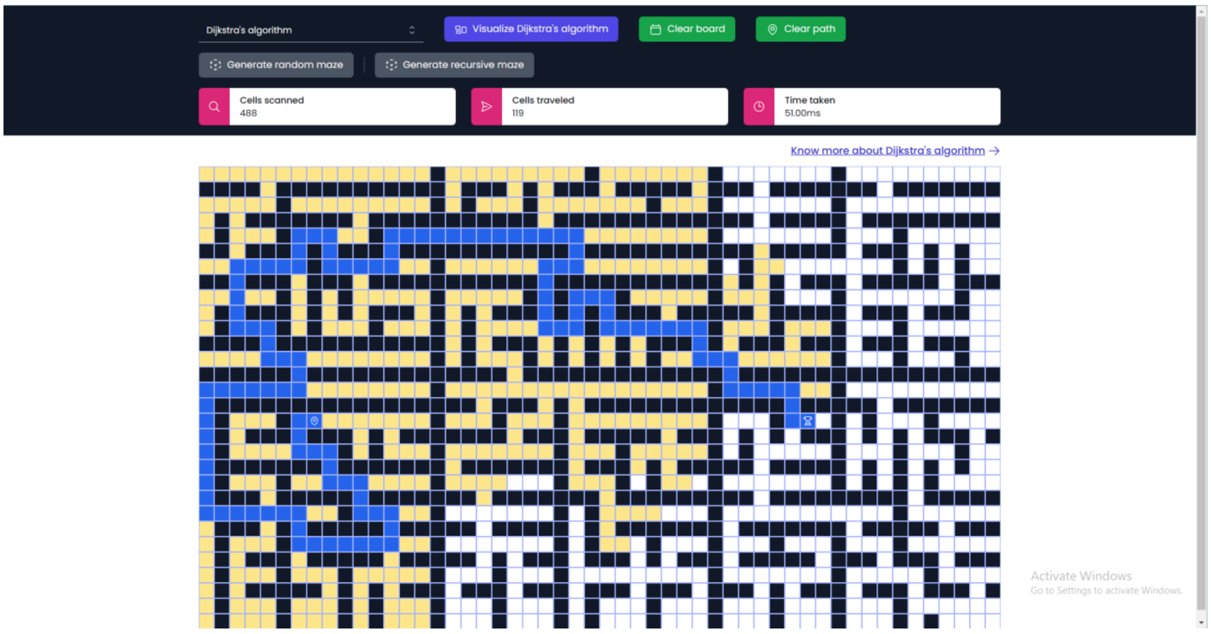
Dijkstra's algorithm
Visualize Dijkstra's algorithm
Clear board
Clear path
Generate random maze
Generate recursive maze
Know more about Dijkstra's algorithm →
Activate Windows
Go to Settings to activate Windows.

Cells scanned
488

Cells traveled
0

Time taken
51.00ms

Shortest path by Dijkastra Algorithm



Dijkstra's algorithm
Visualize Dijkstra's algorithm
Clear board
Clear path
Generate random maze
Generate recursive maze
Know more about Dijkstra's algorithm →
Activate Windows
Go to Settings to activate Windows.

Cells scanned
488

Cells traveled
119

Time taken
51.00ms

12. CONCLUSION FUTURE SCOPE

Conclusion:

In conclusion, the pathfinding visualizer project has successfully achieved its objectives of providing users with a visually interactive tool to understand and compare different pathfinding algorithms. The system's current functionality, combined with its potential for future expansion and enhancements, makes it a valuable resource for both educational and practical purposes. With continuous development and feedback from users, the pathfinding visualizer can become an even more comprehensive and versatile tool in the field of algorithm visualization and exploration.

Future Scope:

The pathfinding visualizer project has been successfully developed, providing users with a powerful tool to visualize various pathfinding algorithms and explore their functionalities. The system allows users to interact with a grid, select algorithms such as Dijkstra's, BFS, and DFS, create random or recursive mazes, and observe the visualization of the pathfinding process. It also provides information such as the time taken, cells traveled, and cells scanned, offering valuable insights into the algorithm's performance.

Throughout the development process, various aspects such as technical feasibility, operational feasibility, and economic feasibility were considered, ensuring that the project was viable and met the desired objectives. The system was implemented using React with Vite, TypeScript, and Tailwind CSS, leveraging the benefits of these technologies to create a responsive and user-friendly web application.

While the current system offers a solid foundation and functionality, there are some limitations and areas for improvement. Currently, it supports a limited set of algorithms, namely Dijkstra's, BFS, and DFS. However, the future scope of the project involves expanding the algorithm options to include additional pathfinding algorithms, such as A* (A-star), Greedy Best-First Search, or Bidirectional Search. This expansion would provide users with a more diverse range of algorithms to experiment with and compare their performances.

Furthermore, the system could benefit from additional features such as user authentication, allowing users to save and load grids, or sharing grid configurations with others. These enhancements would further enhance the user experience and enable collaborative exploration of pathfinding algorithms.

11. REFERENCES

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
https://en.wikipedia.org/wiki/Breadth-first_search
https://en.wikipedia.org/wiki/Depth-first_search
https://www.youtube.com/watch?v=msttfIHHkak&t=1352s
Path finding visualizers by Clément Mihailescu
https://www.javatpoint.com/dijkstras-algorithm
https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
https://www.programiz.com/dsa/graph-bfs