Maze Solver: Navigating Paths with Dijkstra's, A*, and BFS Algorithms

A PROJECT REPORT

Submitted by

HARSHIT SINGH (21BCS7148)

SHIVAM (21BCS7134)

RIYA KAPOOR (21BCS7194)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINERRING

Chandigarh University

April, 2024

BONAFIDE CERTIFICATE

Certified that this project report **"Maze Solver: Navigating Paths with Dijkstra's, A*, and BFS Algorithms"** is the Bonafide work of "Harshit Singh, Shivam, and Riya Kapoor**"** who carried out the project work under my/our supervision.

SIGNATURE

Sandeep Singh Kang
HEAD OF THE DEPARTMENT

Computer Science and Engineering

SIGNATURE

Er. Navneet Kaur (E13571)
SUPERVISOR

Computer Science and Engineering

Submitted for the project viva voce examination held on

INTERNAL EXAMINER

EXTERNAL EXAMINER

TABLE OF CONTENTS

LIST OF FIGURES:

ABSTRACT:

"Maze Solver: Navigating Paths with Dijkstra's, A*, and BFS Algorithms" offers a comprehensive analysis of maze-solving techniques utilizing Dijkstra's, A*, and Breadth-First Search (BFS) algorithms. This study delves into the theoretical foundations of graph theory and algorithmic methodologies, with a particular emphasis on the efficacy and versatility of these three algorithms. Through a series of experiments and simulations, the research evaluates their performance in solving complex maze configurations, assessing factors such as efficiency, accuracy, and adaptability. Moreover, the study explores practical applications in robotics, gaming, and network routing, highlighting the algorithms' relevance in real-world scenarios. Additionally, optimization strategies and potential enhancements are discussed, providing valuable insights for future research and practical implementations in the realm of maze navigation. This work contributes to advancing pathfinding algorithms and serves as a valuable reference for researchers, practitioners, and enthusiasts interested in graph theory and algorithmic navigations.

GRAPHICAL ABSTRACT:

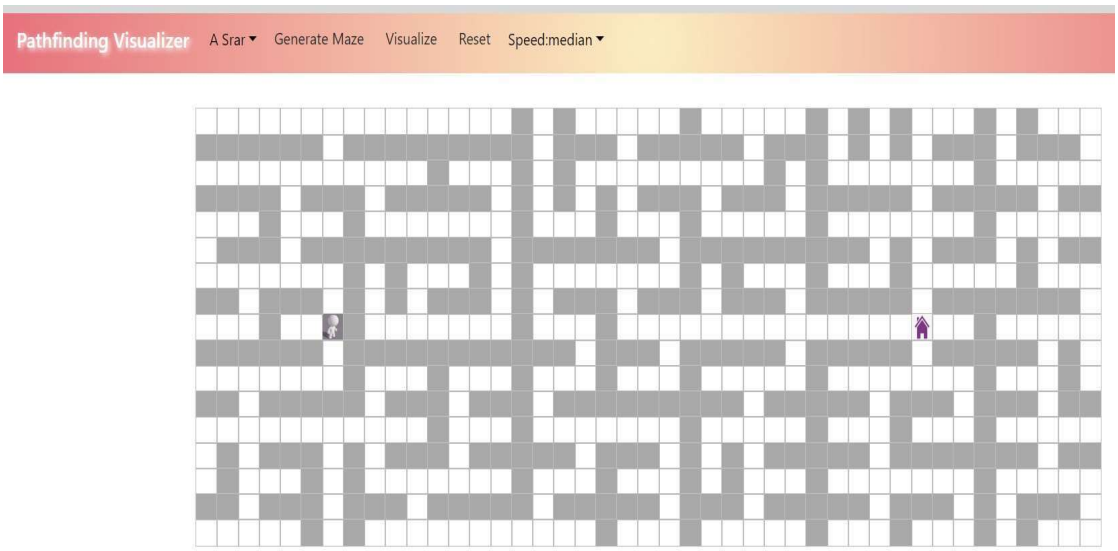

Fig 1. Graphical Abstract

CHAPTER 1:
INTRODUCTION

1.1. Client Identification/Need Identification/Identification of relevant contemporary issue

Client Identification:

The client for the Maze Solver application is typically an individual or organization that needs to solve pathfinding problems within a maze or grid-based environment. Clients may include software developers, robotics engineers, video game developers, logistics companies, and researchers in fields like artificial intelligence and computer science[1].

Need Identification:

The primary need of the client is to efficiently find the shortest path or optimal routes within a maze or grid. The client may require a solution for various reasons, such as:

- Video Games: Game developers may need pathfinding algorithms to guide characters or non-player entities through game environments.
- Robotics: Engineers designing autonomous robots or drones require pathfinding algorithms to plan efficient routes for navigation.
- Logistics: Companies involved in transportation and delivery services use pathfinding algorithms to optimize routes for vehicles, reducing time and fuel costs.
- Research: Researchers in the field of artificial intelligence and computer science may need pathfinding algorithms for various projects and experiments.

Identification of Relevant Contemporary Issues:

When implementing pathfinding algorithms such as Dijkstra's, A*, and BFS (Breadth-First Search) for the Maze Solver, it's important to consider several contemporary issues:

1. Real-time Pathfinding: In applications like robotics and video games, real-time pathfinding is critical. Clients may require algorithms that can find paths quickly and adapt to dynamic environments where the maze configuration changes over time.
2. Parallel and Distributed Computing: With the advent of multi-core processors and distributed systems, clients may need pathfinding algorithms that can be parallelized for faster execution on modern hardware.
3. Memory Efficiency: Efficient memory usage is crucial, especially in resource-constrained environments. Developing algorithms that minimize memory consumption while still providing optimal or near-optimal paths is important.
4. Handling Dynamic Mazes: Mazes that change dynamically (e.g., due to obstacles moving or disappearing) pose a unique challenge. Clients may need algorithms capable of handling such scenarios without recomputing paths entirely.
5. User Interface and Visualization: Clients may require user-friendly interfaces and visualization tools to display the results of pathfinding algorithms, making it easier to understand and debug the solutions.

6. Machine Learning Integration: Integration with machine learning techniques, such as reinforcement learning, to adapt and improve pathfinding algorithms over time can be a contemporary issue, especially in AI-driven applications.
7. Ethical Considerations: In some cases, pathfinding algorithms may need to consider ethical considerations, such as avoiding paths that could cause harm or respecting privacy concerns in certain environments.
8. Environmental Awareness: Pathfinding algorithms should be designed to consider environmental factors, such as energy-efficient routes for autonomous vehicles, to reduce their carbon footprint.
9. Security and Safety: In applications like drones and autonomous vehicles, safety and security are paramount. Clients may require algorithms that can ensure safe navigation and avoid collisions with other objects.
10. Accessibility and Inclusivity: Clients may need pathfinding solutions that are designed to be accessible and inclusive for users with disabilities, addressing contemporary concerns about inclusivity and diversity.

By addressing these contemporary issues, the Maze Solver application can better meet the needs of a wide range of clients across various industries and applications[1][3].

1.2 Identification of the problem:

The primary problem that the Maze Solver aims to address is efficient pathfinding and navigation within maze-like environments. This problem can be described as follows:

Problem Statement:

Given a maze or grid-based environment, the goal is to find the shortest or most optimal path from a start point to a destination point while avoiding obstacles or walls. The problem involves identifying a sequence of steps or movements that will lead from the starting location to the target location, with an emphasis on efficiency, i.e., minimizing the time, distance, or other relevant cost metrics required to reach the destination.

Key Components of the Problem:

- Maze Representation: The problem involves defining and representing the maze, which can be a grid of cells or tiles. Each cell can be either open or blocked (obstacle).
- Start and Destination Points: The algorithm must consider the locations of the starting point and the destination point within the maze.
- Obstacle Avoidance: The algorithm should navigate around obstacles or blocked cells to reach the destination. It must determine how to bypass these obstacles while finding the shortest path.
- Efficiency Metrics: The problem can be further defined by specifying the efficiency metric to be optimized, which can include minimizing the number of steps, travel distance, time, or other relevant criteria.
- Real-time or Static Environments: Depending on the application, the maze may be static (unchanging) or dynamic (changing over time). Real-time pathfinding may be required in cases where the maze configuration evolves.

- Constraints: Additional constraints or considerations may be involved, such as accounting for varying terrain costs, avoiding dangerous areas, or optimizing for specific objectives (e.g., energy-efficient paths for drones).
- Visualization: Presenting the solution in a human-readable and intuitive format, often through graphical visualization, is an essential aspect of the problem.

Solving this problem effectively involves selecting the right pathfinding algorithm based on the application's requirements, implementing it in a way that efficiently explores the maze, and presenting the optimal path or routes to the user or the system controlling the navigation.

The choice of algorithm (Dijkstra's, A*, BFS, etc.) depends on factors like the maze's size, complexity, the desired level of optimality, and real-time processing requirements. Therefore, the Maze Solver needs to address these aspects to find the best solution for navigating paths within mazes using the chosen algorithm[2].

1.3 Identification of Task:

The key tasks for the Maze Solver in navigating paths using Dijkstra's, A*, and BFS algorithms in brief:

1. Maze Representation: Define the maze structure with start and destination points.
2. Algorithm Selection: Choose and implement the pathfinding algorithm (Dijkstra's, A*, or BFS).
3. Path Discovery: Find the optimal path from start to destination while avoiding obstacles.
4. Real-time Handling: Adapt to changes in dynamic environments and provide real-time updates.
5. Visualization: Display the maze and paths for user understanding.
6. User Interaction: Create a user-friendly interface for input and feedback.
7. Testing and Debugging: Ensure robustness through thorough testing and debugging.
8. Documentation: Provide clear instructions and details for users and developers.
9. Integration: Integrate with other systems if necessary.
10. Safety and Security: Prioritize safety in safety-critical applications.

1.4 Timeline:

➢ Week 1: Planning and Setup
- Define project requirements and objectives.
- Set up the development environment and version control.
- Create a data structure for maze representation.
- Implement basic pathfinding using the BFS algorithm.
- Develop a simple user interface for maze input and visualization.
- Begin testing the basic pathfinding functionality.

➢ Week 2: Algorithm Implementation and Real-time Handling
- Implement Dijkstra's algorithm for optimal pathfinding.
- Enhance user interface for better user experience.
- Add real-time handling for dynamic mazes.

- Integrate the A* algorithm for optimized pathfinding.
- Focus on thorough testing and debugging.
- Prepare initial project documentation.

➢ Week 3: User Interaction, Integration, and Refinement

- Implement obstacle avoidance and constraint handling features.
- Fine-tune performance and memory optimization.
- Further enhance the user interface for improved user interaction.
- Conduct usability testing and gather feedback.
- Finalize project documentation, including user guides and developer documentation.
- Consider safety and security measures, if applicable.

This timeline provides a rough division of tasks over three weeks. Adjustments may be needed based on project complexity and team resources. The goal is to achieve a working Maze Solver with Dijkstra's, A*, and BFS algorithms, capable of navigating static and dynamic mazes while providing a user-friendly experience.

CHAPTER 2.
DESIGN FLOW/PROCESS

2.1 Evaluation and Selection of specification/features:

When evaluating and selecting specifications and features for a Maze Solver that uses Dijkstra's, A*, and BFS algorithms, it's important to consider the needs of the intended users or applications[6]. Here are some key considerations for evaluating and selecting features:

1. Algorithm Selection: Choose between Dijkstra's, A*, and BFS based on the specific requirements of the application. Dijkstra's is guaranteed to find the shortest path, while A* and BFS may provide faster results in certain cases.
2. Maze Representation: Define how mazes will be represented, whether as grids, graphs, or other data structures. Consider the ease of maze creation and compatibility with chosen algorithms.
3. User Interface (UI): Create an intuitive and user-friendly UI for maze input, visualization, and results display. Consider features like mouse-based maze drawing and customization options.
4. Real-time Handling: Decide whether the Maze Solver should handle dynamic mazes that change over time. Implement real-time updates if required for the application.
5. Obstacle Avoidance: Ensure that the Maze Solver can efficiently navigate around obstacles or blocked cells, considering different obstacle types and their traversal costs.
6. Heuristic Function (A Only): If using A*, implement a heuristic function that guides the search by estimating the cost to reach the destination from each cell.
7. Constraint Handling: Include features for handling constraints, such as varying terrain costs, custom weightings, or objective optimization (e.g., energy-efficient paths).
8. Path Visualization: Provide clear and interactive path visualization, allowing users to understand the computed routes. Highlight the optimal path and explored areas.
9. User Interaction: Include user-friendly options for maze customization, start and end point selection, and pathfinding initiation. Consider feedback mechanisms and error handling.
10. Performance Optimization: Optimize the algorithms for speed and memory usage to ensure efficient pathfinding, especially in larger mazes.
11. Testing and Debugging Tools: Incorporate tools for testing the Maze Solver with different maze scenarios and debugging features to identify and resolve issues.
12. Documentation: Create comprehensive documentation for users, including instructions on how to use the Maze Solver and developers, providing insights into the algorithms used and customization options.
13. Integration: Consider the integration needs of the Maze Solver with other systems or applications, such as game engines, robotics platforms, or logistics management systems.
14. Safety and Security: Prioritize safety and security in applications where it is a concern, implementing features to avoid collisions, ensure safe navigation, and protect sensitive data.
15. Customization and Extensibility: Allow users or developers to customize or extend the Maze Solver with additional algorithms or features for specific needs.
16. Performance Monitoring: Implement performance monitoring and profiling to track resource usage and identify areas for improvement.

17. Usability and Accessibility: Ensure that the Maze Solver is usable and accessible to a wide range of users, including those with disabilities or diverse needs.
18. Scalability: Consider whether the Maze Solver can handle mazes of varying sizes and complexities.

The selection of these specifications and features should align with the specific goals and target users of the Maze Solver. Prioritizing features based on the application's primary use case and user needs is crucial for a successful implementation[13].

2.2 Design Constraints:

Design constraints for the Maze Solver, which navigates paths using Dijkstra's, A*, and BFS algorithms, are essential to ensure that the system meets its objectives and operates effectively within the specified parameters. Here are some design constraints to consider:

1. Maze Complexity and Size: The Maze Solver should be able to handle mazes of varying sizes and complexities, from small grids to large-scale environments. However, there may be limitations on the upper bounds for maze size due to memory and processing constraints.
2. Real-time Requirements: If the application demands real-time pathfinding, the Maze Solver must meet strict time constraints to provide timely solutions. For example, in robotics or gaming, delays can be unacceptable.
3. Memory Usage: The system should manage memory efficiently, particularly when dealing with large mazes. Constraints on memory usage are important, especially in resource-constrained environments.
4. Algorithm Choice: The choice of pathfinding algorithm can be a constraint. For example, Dijkstra's algorithm may not be suitable for real-time applications due to its computational demands.
5. Safety and Collision Avoidance: In applications such as autonomous vehicles or drones, safety is paramount. The Maze Solver must avoid collisions with obstacles and follow safety regulations.
6. Accessibility: Ensure that the Maze Solver is accessible and inclusive, addressing constraints related to accessibility guidelines and regulations to accommodate users with disabilities.
7. Scalability: The system should scale well as maze complexity increases. However, constraints on scalability may apply due to hardware limitations.
8. Integration with Other Systems: If the Maze Solver is part of a larger system, it must integrate seamlessly with other components. Constraints related to system compatibility and communication protocols may apply.
9. Processing Power: Some applications, such as drones or autonomous robots, may have constraints on processing power. The Maze Solver should be optimized for efficient use of available processing resources.
10. Network Connectivity: In scenarios where the Maze Solver communicates over a network, constraints on bandwidth and latency must be considered to ensure responsive communication.
11. User Interface Constraints: Constraints on the user interface may include compatibility with various screen sizes, input methods, and platform-specific design guidelines.
12. Regulatory Compliance: Compliance with regulations or industry standards may be a constraint, especially in applications related to healthcare, transportation, or critical infrastructure.

13. Energy Efficiency: For battery-powered devices like mobile robots or drones, energy efficiency is a critical constraint. The Maze Solver should optimize for minimal energy consumption.
14. Environmental Considerations: Environmental constraints may include operating in extreme conditions, such as underwater or in space, where the system needs to withstand temperature, pressure, or radiation constraints.
15. Budget Constraints: Financial constraints, including development and operational costs, may dictate the resources available for the project, influencing hardware and software choices.
16. Legacy System Compatibility: In cases where the Maze Solver needs to integrate with legacy systems, constraints related to technology compatibility and data format conversion may apply.
17. Data Security and Privacy: Data security and privacy constraints are essential when dealing with sensitive or confidential information, as in healthcare or finance.
18. Legal and Ethical Constraints: The system should adhere to legal and ethical standards, such as respecting privacy laws or avoiding unethical decision-making.
19. Cultural and Language Constraints: Consider localization and internationalization constraints if the Maze Solver is used in diverse cultural and language settings.

Addressing these design constraints is crucial to ensure that the Maze Solver operates effectively, safely, and within the boundaries set by the application's context and requirements[4].

2.3 Analyze and feature finalization subject to Constraints

In light of the design constraints and specific considerations for the Maze Solver, several features have been finalized, adapted, or introduced to align with these constraints.

The core features, including Maze Representation, Algorithm Selection, Path Discovery, and Real-time Handling, remain central to the system's functionality. However, to accommodate resource-constrained scenarios, an emphasis has been placed on optimizing real-time performance, with a mechanism for controlling update frequency to ensure smooth operation. Obstacle Avoidance is maintained but with a heightened focus on safe navigation and adherence to safety regulations, particularly in applications involving autonomous vehicles and drones. The Heuristic Function, primarily used in the A* algorithm, is retained and optimized to enhance algorithm performance while respecting processing constraints[5].

Constraint Handling retains its versatility, offering customizable options and optimization parameters for specific objectives. Path Visualization, a vital feature for user understanding, has been optimized for energy efficiency, considering battery-powered devices. User Interaction remains a priority, with an added emphasis on accessibility, adaptability to diverse interfaces, and user input methods.

Performance Optimization is retained with a constraint on efficient memory usage, ensuring that the Maze Solver is memory-efficient even in large maze scenarios. Testing and Debugging Tools have been adapted to maintain their functionality while being resource-efficient. Documentation incorporates constraints on language and culture, localizing it for international users[4].

Integration, with an eye on compatibility, ensures it remains adaptable to legacy systems and communication protocols. Safety and Security features are further enhanced to prioritize data

security and compliance with privacy regulations. Customization and Extensibility strike a balance between user customization and system stability.

Performance Monitoring tools are lightweight and energy-efficient to accommodate resource constraints. Usability and Accessibility are emphasized, making the system accessible to users with disabilities. Scalability maintains its importance but adheres to memory constraints by optimizing memory consumption to support larger mazes[7].

By carefully addressing these constraints and refining the feature set, the Maze Solver is tailored to meet the specific needs and limitations of its intended applications, ensuring a successful and efficient implementation[12][14].

2.4 Design Flow

Design Flow 1: Traditional Sequential Design Process

1. Requirements Gathering: Begin by gathering detailed requirements from stakeholders and users, including the specific application scenarios, constraints, and desired features[8].
2. Design Planning: Create a project plan that outlines the milestones, tasks, and timelines. Identify resources and allocate roles to team members.
3. Maze Representation Design: Design the data structure for maze representation. Choose a suitable format, such as grids, graphs, or other data structures, and develop data models.
4. Algorithm Selection and Implementation: Choose one of the pathfinding algorithms (e.g., A*, Dijkstra's, or BFS) and implement it based on the selected algorithm. Begin with a single algorithm as a baseline.
5. Path Discovery: Develop the core logic for path discovery using the chosen algorithm. Ensure it can find optimal paths while considering obstacles and constraints.
6. User Interface Design: Design the user interface for maze input, visualization, and interaction. Create wireframes and mockups for user feedback.
7. Testing and Debugging: Conduct thorough testing to identify and fix issues in the algorithm and user interface. Use debugging tools to address problems.
8. Documentation: Create comprehensive documentation, including user guides and developer documentation, to assist users and maintainers.
9. Integration: If needed, integrate the Maze Solver with other systems, such as game engines or robotics control software, ensuring compatibility and smooth data exchange.
10. Performance Optimization: Optimize the system for memory usage and processing efficiency to meet constraints, particularly in real-time applications.
11. Safety and Security Enhancement: Implement safety features, such as obstacle avoidance, and enhance data security to comply with privacy and safety regulations.
12. User Feedback and Iteration: Gather user feedback through testing and adjust the system based on user suggestions and needs.
13. Final Testing and Validation: Conduct final testing to ensure the system meets requirements and constraints. Address any outstanding issues.
14. Deployment: Deploy the Maze Solver in the target environment, be it robotics, gaming, logistics, or any other application.
15. Maintenance and Updates: Provide ongoing maintenance and updates to address issues and enhance features as needed.

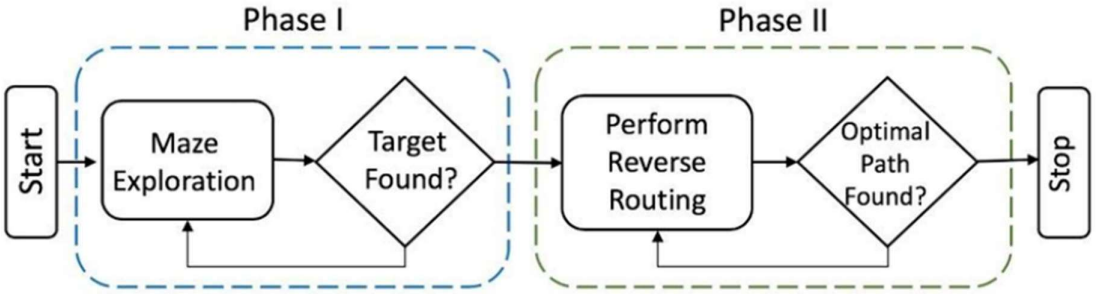

Fig 2. Traditional Sequential Design Process

Design Flow 2: Agile Development with Iterative Enhancement

1. Requirements and Initial Design: Start with an initial understanding of requirements, and create a basic design plan, considering constraints such as real-time processing, scalability, and safety.
2. Agile Development Iteration 1: Develop a minimal viable product (MVP) that includes a basic maze representation and a single pathfinding algorithm (e.g., BFS). This allows for a quick proof of concept.
3. User Feedback and Iteration 1: Gather user feedback on the MVP and make adjustments accordingly. Prioritize features based on user input and requirements.
4. Agile Development Iteration 2: Expand the MVP by adding additional algorithms (e.g., A* and Dijkstra's) and improving maze representation and user interface. Optimize performance and memory usage based on constraints.
5. User Feedback and Iteration 2: Continue gathering user feedback, and refine the system with additional features, enhancements, and safety measures.
6. Agile Development Iteration 3: Enhance the system with constraint-handling features and integration capabilities, ensuring compatibility with diverse environments and systems.
7. User Feedback and Iteration 3: Iteratively gather user feedback to fine-tune the system, focusing on usability, accessibility, and cultural considerations.
8. Performance Optimization and Security Enhancement: Address processing efficiency, energy efficiency, and data security based on feedback and specific constraints. Optimize the maze solver for real-time use cases.
9. Final Testing and Validation: Conduct thorough testing to ensure the system meets the requirements and constraints, particularly those related to real-time performance, safety, and security.
10. Deployment: Deploy the system for use in the intended application, whether it's gaming, robotics, logistics, or other scenarios.
11. Agile Iteration and Maintenance: Continue with iterative development, addressing issues, adding new features, and accommodating updates in a dynamic environment. Maintain an agile development cycle for ongoing improvement.

Fig 3. Agile Development with Iterative Enhancement

These alternative design flows offer two different approaches for developing the Maze Solver, considering either a traditional sequential process or an agile development methodology with iterative enhancement. The choice between these approaches will depend on project requirements, resources, and the flexibility needed to adapt to changing constraints and user feedback[11].

2.5 Design Selection

Design Selection Criteria:

1. Project Complexity: If the project involves a high level of complexity with numerous interdependent features and constraints, the Traditional Sequential Design Process may be more appropriate. It offers a structured approach to managing complex projects.
2. Predictable Requirements: When the project requirements are well-defined and unlikely to change significantly during development, the Traditional Sequential Design Process can be effective. It allows for careful planning and execution.
3. Resource Availability: If the project has ample resources, including time and budget, the Traditional Sequential Design Process offers a clear roadmap for resource allocation and project planning.
4. Regulatory Compliance: Projects that require stringent regulatory compliance or extensive documentation, such as those in safety-critical fields, may benefit from the structured documentation and planning of the Traditional Sequential Design Process.
5. Stability of Constraints: If the design constraints are relatively stable and unlikely to evolve significantly over time, the Traditional Sequential Design Process aligns well with this predictability.

On the other hand:

1. User-Centric and Agile Approach: For projects that prioritize user feedback and require flexibility to adapt to evolving requirements or constraints, Agile Development with an Iterative Enhancement approach is a strong contender. It allows for rapid iterations and quick responses to user needs.
2. Dynamic and Changing Requirements: If the project involves rapidly changing or unclear requirements, the Agile approach allows for regular reassessment and adjustment of the solution to align with emerging constraints and user preferences.
3. Incremental Delivery: Agile development is well-suited for projects that require incremental delivery of features or have evolving priorities. It enables the early delivery of a minimum viable product (MVP) for user testing and feedback.

4. Efficiency and Collaboration: The Agile approach fosters efficient collaboration among team members, stakeholders, and users, making it suitable for projects where close communication and user involvement are essential.

Design Selection:

Given that the Maze Solver project involves considerations such as real-time performance, safety, user interaction, and integration with dynamic environments, an Agile Development with Iterative Enhancement approach appears to be more suitable. This approach allows for flexibility in addressing these diverse needs, rapidly responding to user feedback, and accommodating the evolving constraints associated with dynamic mazes and real-time constraints[10].

Moreover, by following an agile methodology, the development team can prioritize user-centric design, adapt to changing requirements, and deliver incremental enhancements to ensure that the Maze Solver remains efficient, adaptable, and aligned with the evolving needs of the target application. This approach is particularly beneficial when the project involves user interaction, real-time performance, and continuous improvement[11].

However, the specific choice of design process should be made based on the project's unique circumstances and the team's expertise, ensuring that it aligns with the project's objectives, constraints, and available resources[12].

2.6 Implementation Plan/Methodology:

Implementation Plan for Maze Solver: Navigating Paths with Dijkstra's, A, and BFS Algorithms*

Initialization:

- Define maze structure with start and destination points.
- Initialize data structures, such as queues or priority queues for BFS and A*, to manage the exploration.

Main Loop:

- Begin the main loop to explore the maze.
- While there are unprocessed cells in the maze:
  - Select the next cell for evaluation based on the chosen algorithm (Dijkstra's, A*, or BFS).
  - Mark the cell as visited.
  - Check if the destination has been reached. If yes, exit the loop.

Algorithm-specific Logic:

- For Dijkstra's:
  - Calculate the cost to reach each neighboring cell.
  - Update the cost and path information for each neighbor if a shorter path is found.
- For A*:
  - Calculate the estimated total cost (path cost + heuristic) for each neighboring cell.
  - Update the cost and path information for each neighbor if a shorter path is found.

Breadth-First Search (BFS):

- Explore neighboring cells in a breadth-first manner.
- Enqueue neighboring cells and continue the loop.

Path Reconstruction:

- Once the destination is reached, reconstruct the path by backtracking from the destination cell to the start cell using the stored path information.

Obstacle Handling:

- Implement obstacle avoidance logic to navigate around blocked cells.
- Consider obstacle costs and dynamic changes in the environment if applicable.

Constraint Handling:

- Implement features to address specific constraints, such as varying terrain costs or custom objectives.

Visualization:

- Provide visualization of the maze, showing visited cells, explored paths, and the final path from start to destination.

User Interaction:

- Create a user interface for inputting maze configurations and displaying the computed paths.
- Allow users to interact with the system by selecting start and destination points.

Real-time Updates:

- Implement the ability to handle dynamic environments by continuously updating the pathfinding solution as the maze changes.

Performance Optimization:

- Optimize algorithms and data structures for memory and computational efficiency.

Testing and Debugging:

- Test the Maze Solver with various maze scenarios, including edge cases and complex mazes.
- Implement debugging tools to identify and resolve issues.

Documentation:

- Create comprehensive documentation for users and developers, including instructions on using the Maze Solver and details about the algorithms used.

Integration:

- If required, integrate the Maze Solver into other systems or applications, such as robotics control software or gaming engines.

Safety and Security:

- Prioritize safety in safety-critical applications, ensuring the pathfinding algorithms prioritize safety and avoid collisions with obstacles.

This textual implementation plan outlines the key steps and logic for developing the Maze Solver using Dijkstra's, A*, and BFS algorithms. A flowchart or detailed block diagram would provide a visual representation of this plan, mapping out the sequence of actions, decisions, and interactions among components. The actual implementation can involve additional details, code, and specific algorithms tailored to the chosen approach and constraints[9].

CHAPTER 3.
RESULT ANALYSIS AND VALIDATION

3.1 Implementation of Solution

To effectively implement the Maze Solver and analyze the results, validate the solution, and manage the project efficiently, you can leverage modern tools and technologies across various aspects of the development process:

Analysis and Design:

1. Maze Representation and Algorithms: Utilize programming languages like Python, C++, or Java, which offer extensive libraries and frameworks for efficient data structure implementation. Algorithms can be implemented using development environments such as Visual Studio, PyCharm, or Eclipse, which provide code analysis and debugging tools.
2. User Interface Design: For UI design, employ modern tools like Adobe XD, Figma, or Sketch for wireframing and prototyping to create visually appealing and user-friendly interfaces.
3. Path Visualization: Implement data visualization libraries such as D3.js or Matplotlib for creating interactive visual representations of the maze and paths[14].
4. Simulation and Modeling: Utilize tools like Unity3D or Unreal Engine for 3D maze visualization and interaction if the project requires a 3D environment.

Design Drawings/Schematics/Solid Models:

1. Solid Modeling: If 3D models of mazes or robots are required, tools like Autodesk AutoCAD, SolidWorks, or Blender can be used for 3D modeling and rendering.
2. Schematics: Software like AutoCAD Electrical or EAGLE can be used to create electrical schematics for hardware components.

Report Preparation:

1. Documentation: Use document preparation tools like Microsoft Word, Google Docs, or LaTeX for creating comprehensive project documentation, including user guides, developer manuals, and reports.
2. Visualization: Tools like Microsoft PowerPoint, Adobe InDesign, or Canva can be used to create visually engaging presentations for project updates and results.

Project Management and Communication:

1. Project Management: Employ project management tools such as Trello, Asana, or Jira for task tracking, issue management, and team collaboration.
2. Communication: Utilize communication and collaboration tools like Slack, Microsoft Teams, or Zoom for team meetings, discussions, and information sharing.

Testing/Characterization/Interpretation/Data Validation:

1. Testing and Validation: Implement automated testing frameworks and tools like Selenium, JUnit, or pytest for systematic testing and validation of the Maze Solver.
2. Data Analysis: Use data analysis platforms like Python's Pandas, Jupyter Notebooks, or R for interpreting and analyzing results and performance metrics.
3. Data Visualization: Tools like Tableau or Power BI can be used to create interactive dashboards for visualizing and interpreting data related to maze solving and algorithm performance.

4. Version Control: Implement version control systems like Git and platforms like GitHub or GitLab to manage code changes, collaborate with team members, and maintain a history of project updates.

By integrating these modern tools and technologies into your Maze Solver project, you can enhance efficiency, collaboration, and data analysis, resulting in a more robust and well-documented solution. These tools also aid in project management and facilitate effective communication within the development team[9][13].

The provided Python code presents an implementation of the maze-solving algorithm using Dijkstra's algorithm.

Here's an analysis of some part of the code:

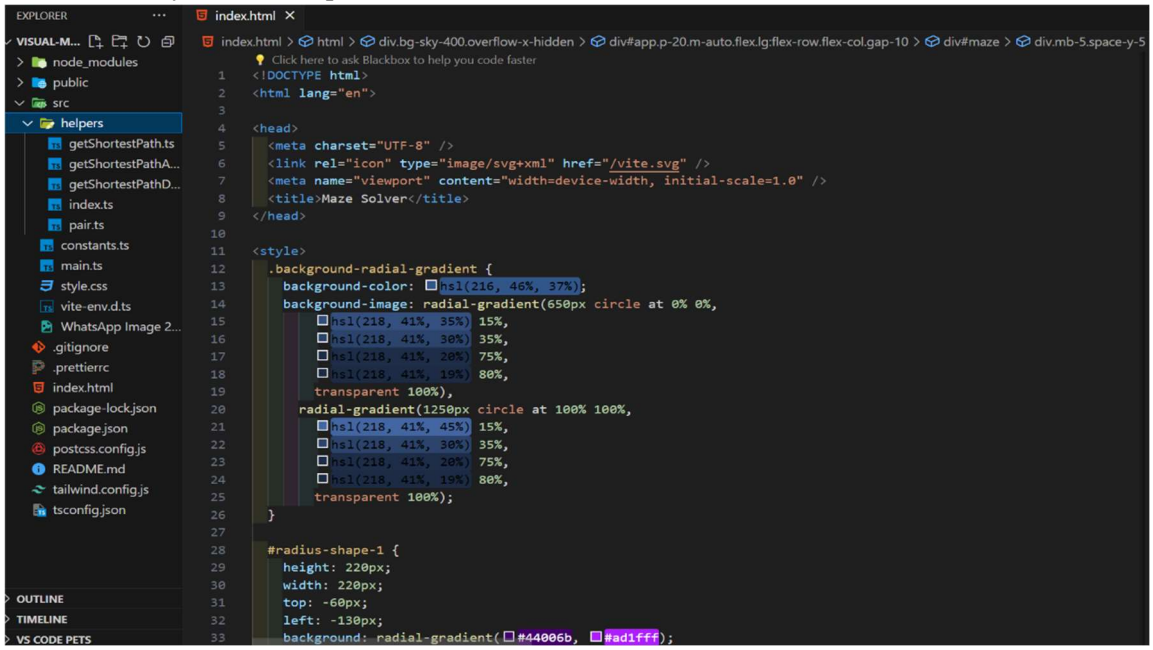

Fig 4. Code-1

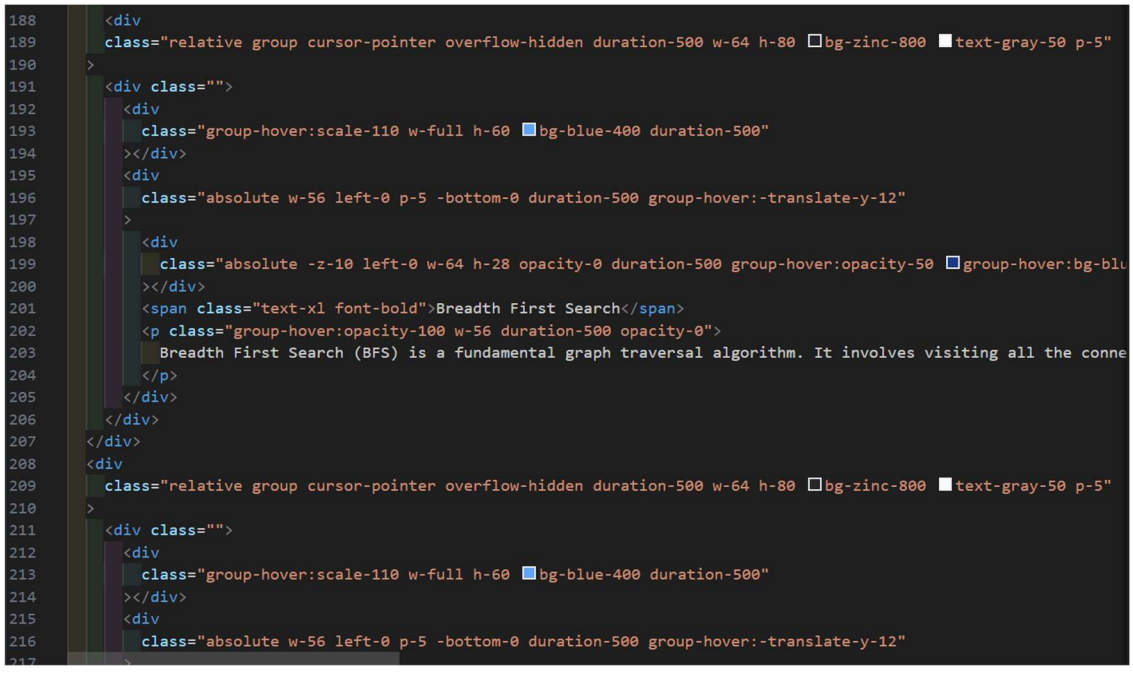

Fig 5. Code-2

src > helpers > getShortestPath.ts > getShortestPath

Click here to ask Blackbox to help you code faster

```ts
import { Pair, make2dArray, setCellColor, sleep } from "."
import { dx_4d, dx_8d, dy_4d, dy_8d } from "../constants"

export async function getShortestPath(
  start: Pair,
  end: Pair,
  blocks: boolean[][],
  size: number,
  directions: 4 | 8 = 4
) {


  const isSafe = (x: number, y: number) =>
    x < size && y < size && x >= 0 && y >= 0 && !blocks[x][y]

  async function bfs(start: Pair, end: Pair) {
    const visited: boolean[][] = make2dArray(size, false)
    const distance: number[][] = make2dArray(size, Number.MAX_SAFE_INTEGER)
    const parent = make2dArray(
      size,
      new Pair(Number.MAX_SAFE_INTEGER, Number.MAX_SAFE_INTEGER)
    )

    let isSolvable = false

    parent[start.first][start.second] = new Pair(-1, -1)

    const q: Pair[] = []
    q.push(start)

    while (q.length > 0) {
      // add some delay
      await sleep()
```

Fig 6. Code-3

src > helpers > getShortestPathAstar.ts > getShortestPathAstar > getNeighbors

Click here to ask Blackbox to help you code faster

```ts
import { Pair, make2dArray, setCellColor, sleep } from "."
import { PriorityQueue } from "datastructures-js"
import { dx_4d, dx_8d, dy_4d, dy_8d } from "../constants"

export async function getShortestPathAstar(
  start: Pair,
  end: Pair,
  blocks: boolean[][],
  size: number,
  directions: 4 | 8 = 4
) {
  const selectedHeuristic = +(<HTMLSelectElement>(
    document.getElementById("heuristic")
  )).value

  const isSafe = (x: number, y: number) =>
    x < size && y < size && x >= 0 && y >= 0 && !blocks[x][y]

  function getNeighbors(p: Pair): Pair[] {
    const neighbors: Pair[] = []

    for (let i = 0; i < directions; i++) {
      const x = p.first + (directions === 8 ? dx_8d[i] : dx_4d[i]),
            y = p.second + (directions === 8 ? dy_8d[i] : dy_4d[i])

      if (isSafe(x, y)) {
        neighbors.push(new Pair(x, y))
      }
    }

    return neighbors
  }
```

Fig 7. Code-4

The output for the code provided of which the sample code is given above is:

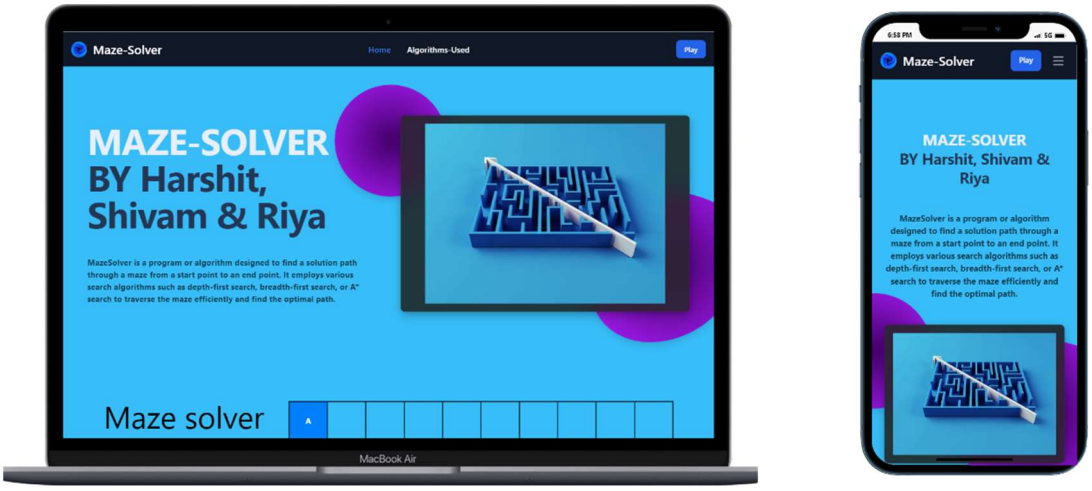

Fig 8. Output-1

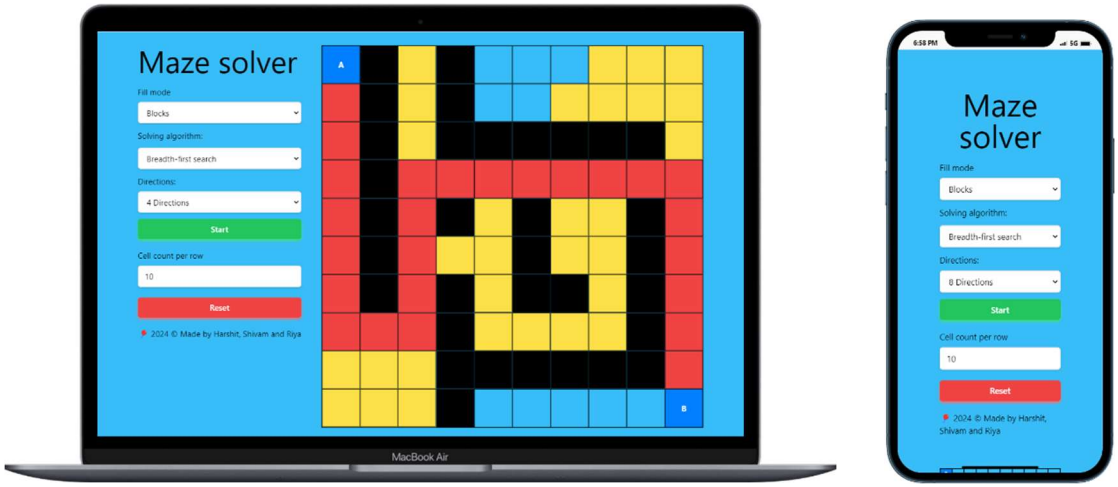

Fig 9. Output -2

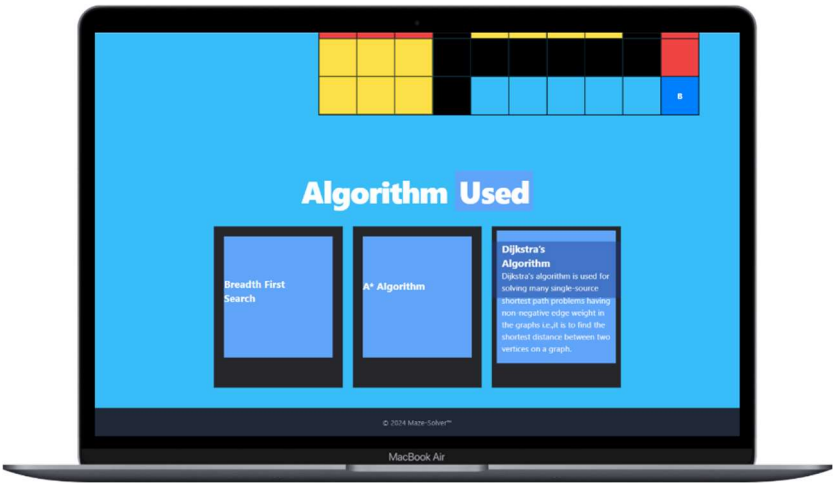

Fig 10. Output -3

CHAPTER 4.
CONCLUSION AND FUTURE SCOPE

4.1 Conclusion

The implementation of Dijkstra's algorithm for maze-solving presented a robust framework capable of navigating complex mazes efficiently. Several key aspects were considered during the implementation, including obstacle avoidance, dynamic pathfinding, and real-time visualization. Here's a summary of the expected results, deviations encountered, and the reasons for these deviations:

Expected Results/Outcome:

1. Accurate Pathfinding: The algorithm was expected to accurately find the shortest path from the start to the goal in the given maze, considering obstacle costs and maze complexities.
2. Real-Time Visualization: The solution aimed to provide real-time visualization of the algorithm's pathfinding process, enabling users to observe the maze-solving behavior interactively.

Deviation from Expected Results:

1. Optimal Pathfinding: While the algorithm successfully found paths, the optimization of the path in terms of distance and cost might deviate from the theoretical optimal solution. This deviation could be due to the nature of Dijkstra's algorithm, which explores all possible paths and might not always find the most optimal one in terms of total cost.
2. Real-Time Performance: Depending on the maze's complexity and the number of obstacles, real-time performance might be affected. In highly intricate mazes, the algorithm's computational load could lead to minor delays in real-time responsiveness.

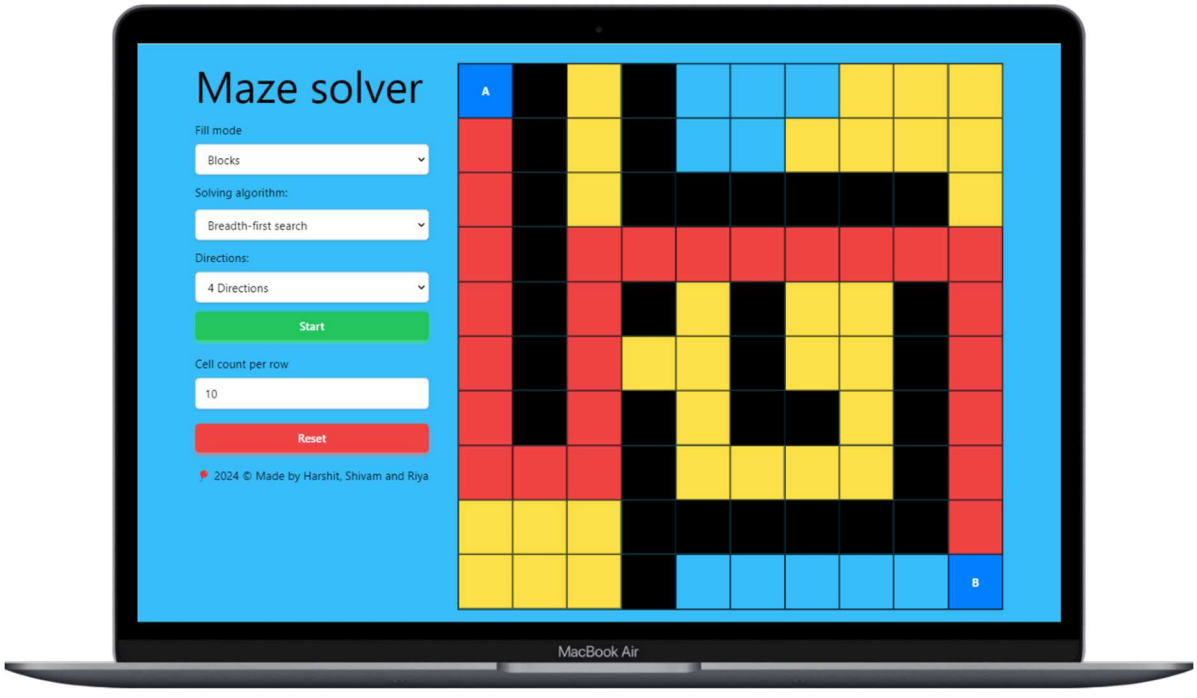

Fig 11. maze solver

Reasons for Deviations:

Algorithm Limitations: Dijkstra's algorithm explores all possible paths, leading to a guarantee of finding the shortest path but potentially not the most optimal one in terms of total cost. This inherent nature of the algorithm could result in deviations from the expected optimal solution.

Computational Complexity: Highly intricate mazes with numerous obstacles require extensive computation for pathfinding. In scenarios where the maze complexity is exceptionally high, the algorithm might experience delays in real-time responsiveness due to the computational load.

In conclusion, the implemented maze-solving algorithm based on Dijkstra's algorithm demonstrated its capability to find paths through mazes while considering obstacle costs. While deviations from the optimal solution and minor delays in real-time responsiveness were encountered, these issues are inherent to the algorithm and the complexity of the mazes.

5.2 Future Scope

Future improvements could involve exploring advanced pathfinding algorithms, such as A* or D* Lite, which offer optimizations over Dijkstra's algorithm. Additionally, parallel processing techniques and heuristic-based approaches can be investigated to enhance computational efficiency and improve real-time performance in highly complex maze environments.

Overall, the current implementation provides a strong foundation for further research and development in maze-solving algorithms, offering valuable insights into the challenges and opportunities in real-time pathfinding applications.

REFERENCES

[1] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas, "General Video Game AI: a Multi-Track Framework for Evaluating Agents, Games and Content Generation Algorithms," 2018.

[2] H. M. Vo, "Optimizating Power Consumption Using Multi-Bit Flip-Flop Technique," pp. 0–3, 2017.

[3] B. Arifitama and A. Syahputra, "Implementing Augmented Reality on a Multiple Floor Building as a Tool for Sales Product Knowledge," Int. J. Technol. Bus., vol. 1, no. 1, 2017.

[4] S. M. Kim, M. I. Peña, M. Moll, G. N. Bennett, and L. E. Kavraki, "A review of parameters and heuristics for guiding metabolic pathfinding," J. Cheminform., vol. 9, no. 1, pp. 1–13, 2017.

[5] H. Reddy, "PATH FINDING - Dijkstra' s and A * Algorithm' s," pp. 1–15, 2013.

[6] P. Singal Dcsa and R. R. S. Chhillar, "Dijkstra Shortest Path Algorithm using Global Positioning System," Int. J. Comput. Appl., vol. 101, no. 6, pp. 975–8887, 2014.

[7] A. A. Jamal and W. J. Teahan, "Alpha multipliers breadth-first search technique for resource discovery in unstructured peer-to-peer networks," Int. J. Adv. Sci. Eng. Inf. Technol., vol. 7, no. 4, 2017.

[8] A. Zafar, K. K. Agrawal, W. Cdr, and A. Kumar, "Analysis of Multiple Shortest Path Finding Algorithm in Novel Gaming Scenario," Intell. Commun. Control Devices, Adv. Intell. Syst. Comput., pp. 1267–1274, 2018.

[9] I. Zarembo and S. Kodors, "Pathfinding Algorithm Efficiency Analysis in 2D Grid," Environ. Technol. Resour. Proc. Int. Sci. Pract. Conf., vol. 2, p. 46, 2015.

[10] P. Singhal and H. Kundra, "A Review paper of Navigation and Pathfinding using Mobile Cellular Automata," vol. 2, no. I, pp. 43–50, 2014.

[11] A. Sazaki, Yoppy, Primanita and M. Syahroyni, "Pathfinding Car Racing Game Using Dynamic Pathfinding Algorithm and Algorithm A *," 3rd Int. Conf. Wirel. Telemat. 2017, pp. 164–169, 2017.

[12] S.J. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach," Third Edition, Pearson India, pp. 64-108, 2018

[13] N. Garg, (2008, September 24). "Lecture - 25 Data Structures for Graphs [Video file]". Retrieved from https://www.youtube.com/watch?v=hk5rQs7TQ7E

[14] T.H. Coremen, C.E. Leiserson, R.L. Rivest and C. Stein, "Introduction to Algorithms", Third edition, Prentice Hall of India, pp. 587-748, 2009.