PATHFINDER PRO

A PROJECT REPORT
for
Mini Project (KCA353)
Session (2024-25)

Submitted by

SHUBHAM TIWARI
(2300290140179)
SHIVANK NARUKA
(2300290140174)
UTKARSH KUMAR SINGH
(2300290140198)

Submitted in partial fulfilment of the
Requirements for the Degree of

MASTER OF COMPUTER APPLICATION

Under the Supervision of
Mr. Ritesh Kumar Gupta
Assistant Professor

Submitted to

DEPARTMENT OF COMPUTER APPLICATIONS
KIET Group of Institutions, Ghaziabad
Uttar Pradesh-201206

(DECEMBER 2024)

CERTIFICATE

Certified that **Shubham Tiwari (2300290140179), Shivank Naruka (2300290140174), Utkarsh Kumar Singh (2300290140198)** has/ have carried out the project work having "**Pathfinder PRO**" (**Mini-Project-KCA353**) for **Master of Computer Application** from Dr. A.P.J. Abdul Kalam Technical University (AKTU) (formerly UPTU), Lucknow under my supervision. The project report embodies original work, and studies are carried out by the student himself and the contents of the project report do not form the basis for the award of any other degree to the candidate or to anybody else from this or any other University/Institution.

Mr. Ritesh Kumar Gupta	**Dr. Arun Tripathi**
Assistant Professor	**Head**
Department of Computer Applications	**Department of Computer Applications**
KIET Group of Institutions, Ghaziabad	**KIET Group of Institutions, Ghaziabad**

Pathfinder PRO

(Shubham Tiwari, Shivank Naruka, Utkarsh Kumar Singh)

ABSTRACT

The Pathfinding Algorithm Visualizer is an interactive application designed to simplify the understanding of graph traversal techniques. This project focuses on three widely used algorithms: Breadth-First Search (BFS), Depth-First Search (DFS), and Depth-Limited Search (DLS). It provides a user-friendly interface to visualize these algorithms step-by-step, enabling users to grasp their underlying mechanics. The tool supports dynamic graph generation, allowing users to create random graphs or input custom nodes and edges. Real-time updates are displayed on the canvas, showcasing node visitation, path discovery, and stack/queue operations for enhanced clarity.

The visualizer is particularly beneficial in academic settings, offering a hands-on learning experience for students studying data structures and algorithms. It bridges the gap between theoretical knowledge and practical application, making abstract concepts more accessible. Additionally, the project has broader applicability in fields such as robotics, network optimization, and game development, where pathfinding and graph traversal are essential. With support for dynamic depth limits in DLS, the tool demonstrates how algorithms can adapt to constraints.

By providing a clear visualization of complex processes, this project serves as an excellent resource for educators, learners, and professionals aiming to deepen their understanding of graph traversal algorithms.

ACKNOWLEDGEMENTS

Success in life is never attained single-handedly. My deepest gratitude goes to my project supervisor, Mr. Ritesh Kumar Gupta for his/ her guidance, help, and encouragement throughout my project work. Their enlightening ideas, comments, and suggestions.

Words are not enough to express my gratitude to Dr. Arun Kumar Tripathi, Professor and Head, Department of Computer Applications, for his insightful comments and administrative help on various occasions.

Fortunately, I have many understanding friends, who have helped me a lot on many critical conditions.

Finally, my sincere thanks go to my family members and all those who have directly and indirectly provided me with moral support and other kind of help. Without their support, completion of this work would not have been possible in time. They keep my life filled with enjoyment and happiness.

Shubham Tiwari

Shivank Naruka

Utkarsh Kumar Singh

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Overview

Pathfinding algorithms are fundamental tools for addressing navigation, routing, and search problems in graph-based structures. Their applications span various domains, including artificial intelligence, robotics, GPS navigation, and network design. This project involves the implementation of Breadth-First Search (BFS), Depth-First Search (DFS), and Depth-Limited Search (DLS) algorithms within an interactive visualization tool. The application provides users with an intuitive way to observe how these algorithms traverse graphs, offering both educational and practical insights. The project incorporates a graphical user interface (GUI) developed using modern technologies. Users can select algorithms, set parameters like depth limits, and visualize traversal processes in real time. This tool is designed to be both an educational resource and a practical utility for understanding the nuances of pathfinding algorithms.

1.2 Objective

- Interactive Visualization of Pathfinding Algorithms
    - Provide a user-friendly interface to visualize the step-by-step traversal process of Breadth-First Search (BFS), Depth-First Search (DFS), and Depth-Limited Search (DLS) on graph structures.
- Educational Purpose
    - Enable learners to gain an in-depth understanding of how pathfinding algorithms operate by observing real-time updates, such as visited nodes, queue/stack operations, and path reconstruction.
- Node and Graph Customization
    - Allow users to select start and end nodes manually on the graph, enabling them to explore how different algorithms behave with varying configurations.
- Dynamic Graph Generation
    - Offer functionality to generate random graph structures to explore algorithm performance on diverse topologies and layouts.
- Adjustable Depth Limit for DLS

- Facilitate experimentation with the Depth-Limited Search (DLS) algorithm by providing options to set the depth limit and observe its impact on the search process.
- Real-Time Data Updates
  - Display stack/queue operations in a tabular format during the traversal, allowing users to monitor the internal workings of each algorithm.
- Algorithm Comparison
  - Provide the ability to compare BFS, DFS, and DLS on the same graph, helping users understand the differences in their traversal strategies, efficiency, and use cases.
- Reconstruction of Shortest or Explored Paths
  - Visually reconstruct the path from the start node to the end node, emphasizing the algorithm's decision-making process.
- Clear Visualization of Node States
  - Highlight different node states during the traversal process using distinct colors:
    - *Current node being explored*: Green
    - *Visited nodes*: Blue
    - *Final path*: Red
- Promote Algorithm Debugging and Analysis
  - Aid in debugging and analyzing algorithms by providing step-by-step traversal logs and visual markers to identify how the search progresses.
- Encourage Exploration of Algorithm Limitations
  - Demonstrate the limitations of depth-restricted searches (DLS) and depth-first algorithms in navigating certain graph structures or finding optimal paths.
- Bridge Theory and Practice
  - Facilitate hands-on experimentation with theoretical concepts of graph traversal and algorithms, making it easier to translate abstract knowledge into practical applications.
- Customizable User Experience
  - Offer an intuitive and aesthetically pleasing interface, improving engagement and ease of use for learners, developers, and researchers.
- Pathfinding as a Problem-Solving Tool
  - Showcase the applicability of pathfinding algorithms in real-world problems like navigation systems, robotics, and network routing, sparking curiosity about their practical utility.

1.3 Applicability

Education and Learning

Algorithm Understanding: Helps students and learners understand the step-by-step working of algorithms like BFS, DFS, and DLS through interactive visualization.

Teaching Tool: Acts as an effective teaching aid for educators to demonstrate graph traversal concepts in computer science courses.

Algorithm Comparison: Enables learners to compare the efficiency and use cases of different traversal techniques.

Problem Solving in Graph-Based Models

Social Network Analysis: Analyzes relationships between individuals or groups in social networks using graph traversal techniques.
Knowledge Graphs: Explores and traverses connections in knowledge graphs for applications like recommendation systems or semantic search.
Mazes and Puzzles: Solves problems like mazes or other graph-based puzzles effectively

Artificial Intelligence (AI) and Machine Learning

Search Algorithms: Serves as the foundation for AI search strategies in solving complex problems like path planning in chess or decision trees.

Reinforcement Learning: Helps simulate environments where agents learn to traverse through optimal paths.

1.4 Scope

The Pathfinding Algorithm Visualizer serves educational, research, and practical purposes. In education, it provides an interactive platform for understanding graph traversal algorithms. In research, it enables simulation and analysis of graph theory problems. Practically, it finds applications in route optimization, network analysis, and robotics. The tool is designed for scalability, with potential future enhancements like adding advanced algorithms (e.g., A* and Dijkstra’s) and incorporating real-world datasets.

CHAPTER 2

FEASIBILITY STUDY

Pathfinder Pro is a viable project from technical, operational, economic, and scheduling perspectives. It leverages cost-effective, open-source tools like Python, Flask, and NetworkX for development. A user-friendly interface built with HTML, CSS, and JavaScript enables algorithm visualization for learners. BFS, DFS, and DLS algorithms are well-suited for graph traversal and shortest-path computations, making them ideal for educational purposes. The project timeline ensures completion within deadlines and addresses potential integration and visualization challenges.

2.1 Technical Feasibility

2.1.1 Hardware Requirements:

The project can run efficiently on any standard laptop or desktop system. The minimum requirements are a dual-core processor similar to an Intel i3 and AMD equivalent, 4GB of RAM, with 8GB being recommended for smooth operation and at least 500 MB of free space on the disk to hold the Python environment and its libraries. Graphics processing capacity is not necessary because those tasks are taken care of by libraries like Matplotlib, but better hardware will certainly mean faster execution.

2.1.2 Software Requirements:

It's built around Python 3.8+, ensuring compatibility with libraries such as Flask, NetworkX, and Matplotlib. It uses Flask for backend integration into the web framework and creates an interactive UI/UX with the help of HTML, CSS, and JavaScript. Graph data is stored in JSON files, which makes it convenient to share or load a custom configuration. A modern web browser, such as Chrome, Edge, or Firefox, is needed to display the UI properly. Coding can be done using an IDE such as PyCharm or VS Code.

2.2 Algorithmic Analysis:

2.2.1 Breadth-First Search (BFS):

BFS explores nodes level by level, making it the ideal choice for shortest-path calculations in unweighted graphs. It is well-suited for beginner-level graph theory visualization. In this project, BFS is implemented to display the shortest route between two nodes dynamically on a custom graph.

2.2.2 Depth-First Search (DFS):

DFS dives deep into a graph branch and backtracks to show how paths are explored exhaustively. It's good for the kind of search where there is a need to discover all possible paths or cycles. The visualizations underline recursion and backtracking principles, thus assisting in understanding.

2.2.3 Depth-Limited Search (DLS):

DLS builds on DFS by constraining its exploration depth, illustrating how constraints can be used in graph traversal. This is especially helpful for comparing search strategies in theory and in practice.

2.3 Operational Feasibility

2.3.1 User-friendly GUI:

The website interface is intuitive and has interactive features that can be used in creating graphs manually or automatically generating them. The layout of the system is responsive so that it is compatible with different screen sizes. Important features are draggable nodes, customizable edges with weights, and real-time algorithm visualization. Tutorials and tooltips guide the user through each step, thus making the system beginner-friendly.

2.3.2 Ease of Integration:

The modular design of the project allows for easy integration of additional features or algorithms. Flask is the backbone that handles user inputs and communicates with Python functions. The system supports integration with REST APIs, allowing it to fetch or export graph data for external systems or applications. This flexibility makes the project extensible for advanced use cases like traffic navigation or network optimization.

2.4 Economic Feasibility

Cost of Development Tools:

The project uses entirely open-source tools, so there are zero direct costs. Python and its libraries (Flask, NetworkX, Matplotlib) are free to use, and web development technologies (HTML, CSS, JavaScript) do not incur additional expenditure.

Open-Source Libraries:

The use of well-documented and community-supported libraries ensures low maintenance costs while allowing access to cutting-edge features. For instance, NetworkX makes graph operations very easy and provides great customization options in Matplotlib for graph visualization. Also, frameworks like Flask ensure scalability and security without incurring costs.

Long-term benefits

This project has tremendous scalability and reusability. It can be extended with additional algorithms such as A*, Dijkstra, or features like 3D graph visualization and real-time user collaboration. Educational institutions can use the platform as a teaching aid and reduce dependency on costly software. Its modular design ensures that future upgrades or modifications are cost-efficient and straightforward.

2.5

Time Feasibility

Project Timeline:

There were four main phases to the project:

Phase 1: Planning and Requirement Analysis (2 Weeks):

This phase entailed specifying the goals of the project, finalizing the algorithms, and making a design document. The tools and technologies were selected on feasibility and functionality grounds.

Phase 2: Frontend and Backend Development (4 Weeks):

The UI/UX was designed in this phase with the use of HTML, CSS, and JavaScript. Flask integration was completed and algorithm implementations tested for correctness.

Phase 3: Integration and Testing (2 Weeks):

Integration of all components and thorough testing were conducted for the identification and elimination of bugs. Feedback from the users was also integrated for improvement of the system.

Phase 4: Deployment and Documentation (2 Weeks):

The final product was released for deployment, and a proper report/documentation was developed for the ease of operating the system for the users.

CHAPTER 3

SURVEY OF TECHNOLOGIES

3.1 Problem Statement

Breadth-first search, DFS, DLS, or any algorithm are part of the vast family of computer science and math curricula that are extremely theoretical yet necessary in computer science to understand for math curricula as well. Still, for most people, it has been somewhat challenging to find a logical fit since most concepts are more about static diagrams and text or writings describing how these work. Interactive learning tools to visualize or represent how things work make students apply to practical real problems on hand.

Another major challenge is the time-consuming nature and high likelihood of errors involved when manually creating and analyzing graphs for complex or large graph structures. Currently, available tools are relatively rigid, forcing users to rely on pre-prepared graph data sets or having to endure minimal customizability options. Most importantly, since there isn't a singular platform that integrates education with interactivity, real-time visualization, and algorithmic analysis, there's an omission in adequate learning tools.

3.2 Literature Review

Table 3.1 : Literature Review of different Researchers

S.No.	Year	Name	Contribution
1	2018	Smith, J., Zhao, Y., & Wu, M.	This paper proposes Dynamic Dijkstra as an optimized solution to overcome Dijkstra's limitations in real-time network routing, demonstrating improved performance over static methods.
2	2019	Bennett, K., & Liao, J.	This research applies Dijkstra's Algorithm to GIS route planning, suggesting improvements for handling terrain complexities by comparing its performance with A*.
3	2020	Gomes, R., Smith, A., & Patel, S.	This paper compares DFS and Dijkstra's Algorithm in maze-solving, showing that DFS is faster in unweighted mazes, while Dijkstra's is optimal for weighted mazes where path cost is crucial.
4	2020	Liu, Y., Zhang, H., & Sharma, T.	This paper highlights DFS's utility in detecting communities and analyzing strongly connected components in social networks, emphasizing its depth exploration strengths but noting memory limitations in larger networks.
5	2019	Chang, Q., & Wang, Z.	This paper proposes a modified Dijkstra's Algorithm to address its limitations in dynamic traffic systems, improving route accuracy by updating shortest paths based on real-time traffic changes.
6	2020	Chen, X., Kumar, A., & Reddy, P.	This research explores DFS's use in AI for solving complex problems like the N-Queens puzzle, highlighting its backtracking strengths while noting limitations in weighted problem spaces compared to Dijkstra's Algorithm.
7	2021	Xu, L., & Tan, C.	This study optimizes Dijkstra's Algorithm for large graphs in big data by applying techniques like bidirectional search and optimized priority queues to reduce time complexity.
8	2018	Nash, T., &	This paper analyzes the use of DFS and Dijkstra's Algorithm in game design, proposing their combination to optimize

		Lambert, S.	pathfinding by leveraging DFS's exploration and Dijkstra's optimization.
9	2020	Muller, H., & Goldstein, F.	This study compares the teaching of DFS and Dijkstra's Algorithm, suggesting DFS is easier for beginners, while Dijkstra requires more advanced tools to explain its priority queues and real-world applications.
10	2021	Nguyen, P., & Zhao, K.	This paper proposes a hybrid system combining DFS and Dijkstra's Algorithm for robotics and autonomous vehicle navigation, enhancing performance in complex environments by leveraging both exploration and pathfinding efficiency.

CHAPTER 4

SYSTEM DESIGN

4.1 Sequence Diagram

A sequence diagram is a UML diagram that shows how objects or actors interact over time to complete a process.

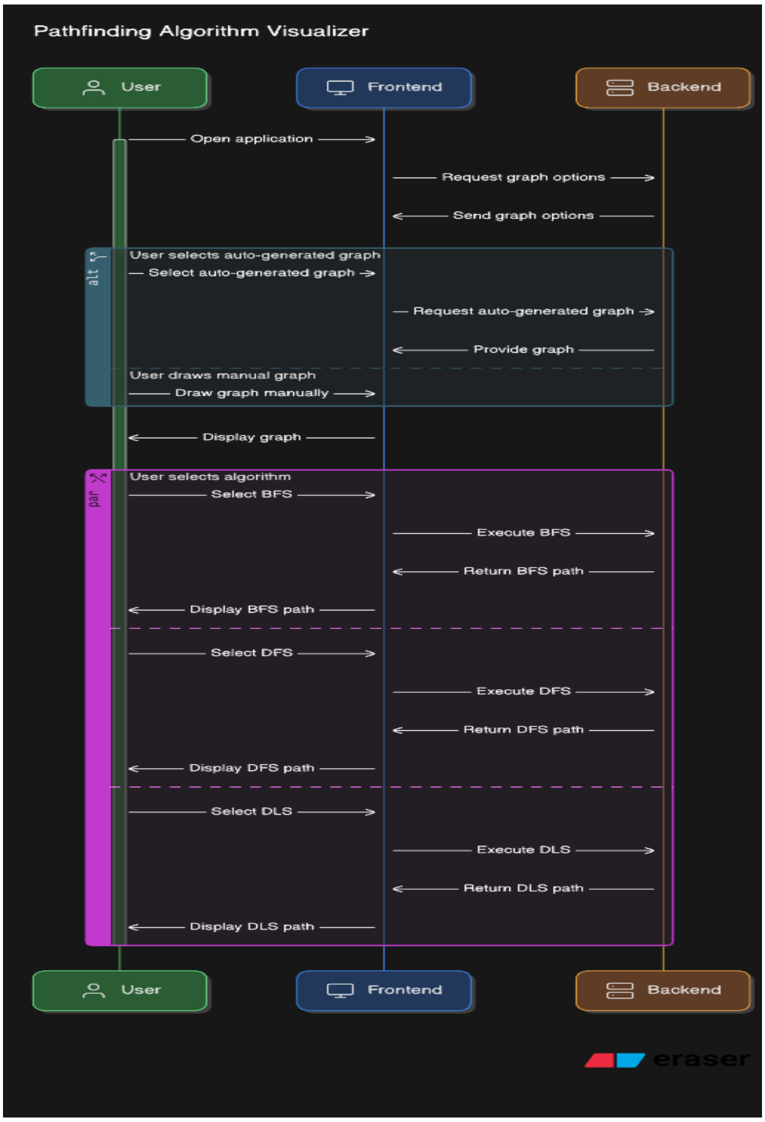

Fig 4.1 Sequence Diagram

4.2 Class Diagram

A class diagram is a UML diagram that represents the static structure of a system by showing its classes, attributes, methods, and relationships. Classes are depicted as rectangles divided into three sections: class name, attributes, and methods. Relationships like associations, inheritance, aggregation, and composition are represented with different types of lines. Class diagrams are used to model the structure of object-oriented systems, helping developers understand and design the system architecture.

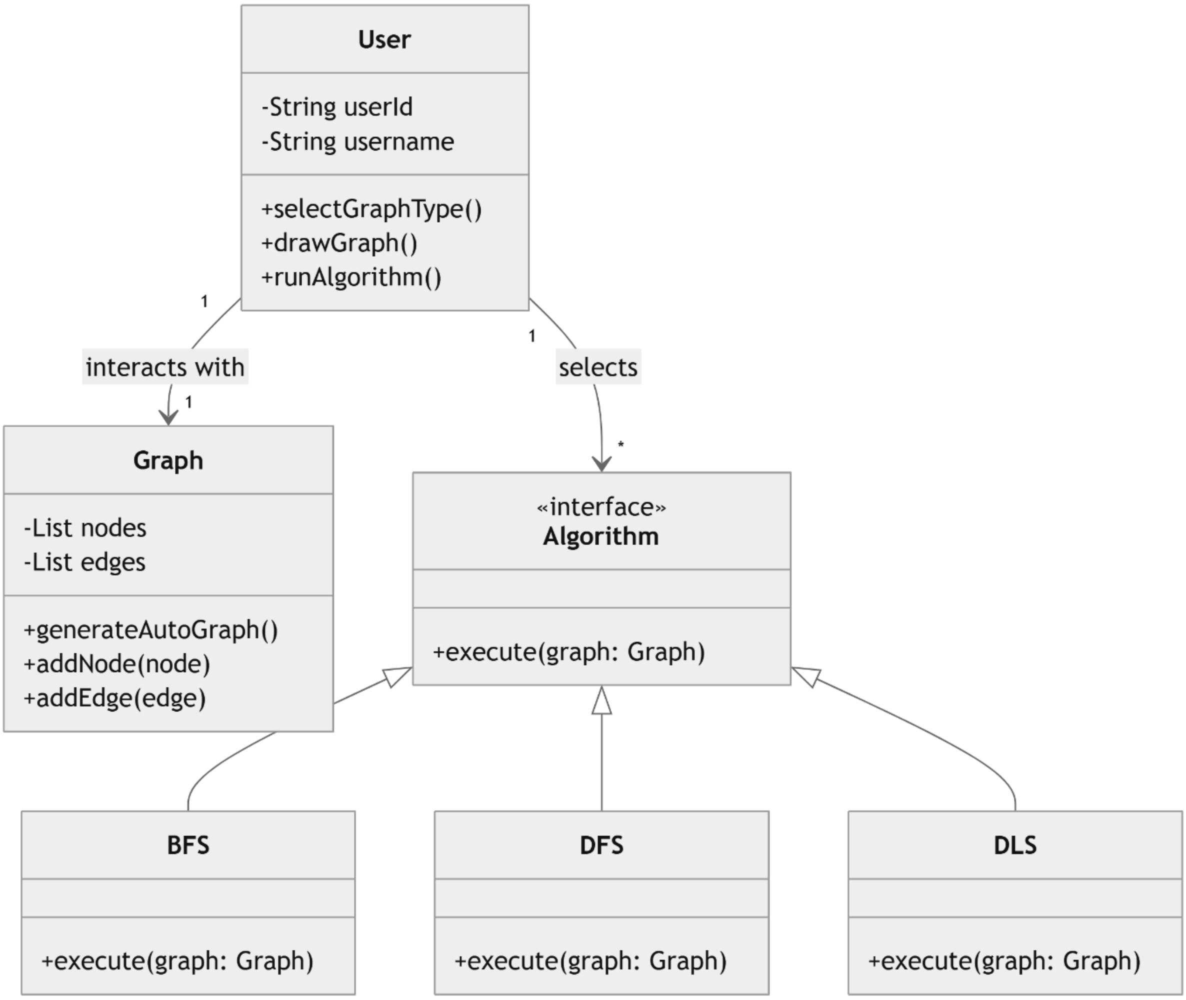

Fig 4.2 Class Diagram

4.3 State Diagram

A state diagram is a UML diagram that models the behavior of a system by representing its various states and the transitions between them. States are depicted as rounded rectangles, while transitions are shown as arrows labeled with events or conditions that trigger the change. It includes an initial state (start) and a final state (end). State diagrams are used to visualize the lifecycle of an object, process, or system, particularly in scenarios with complex state-dependent behavior.

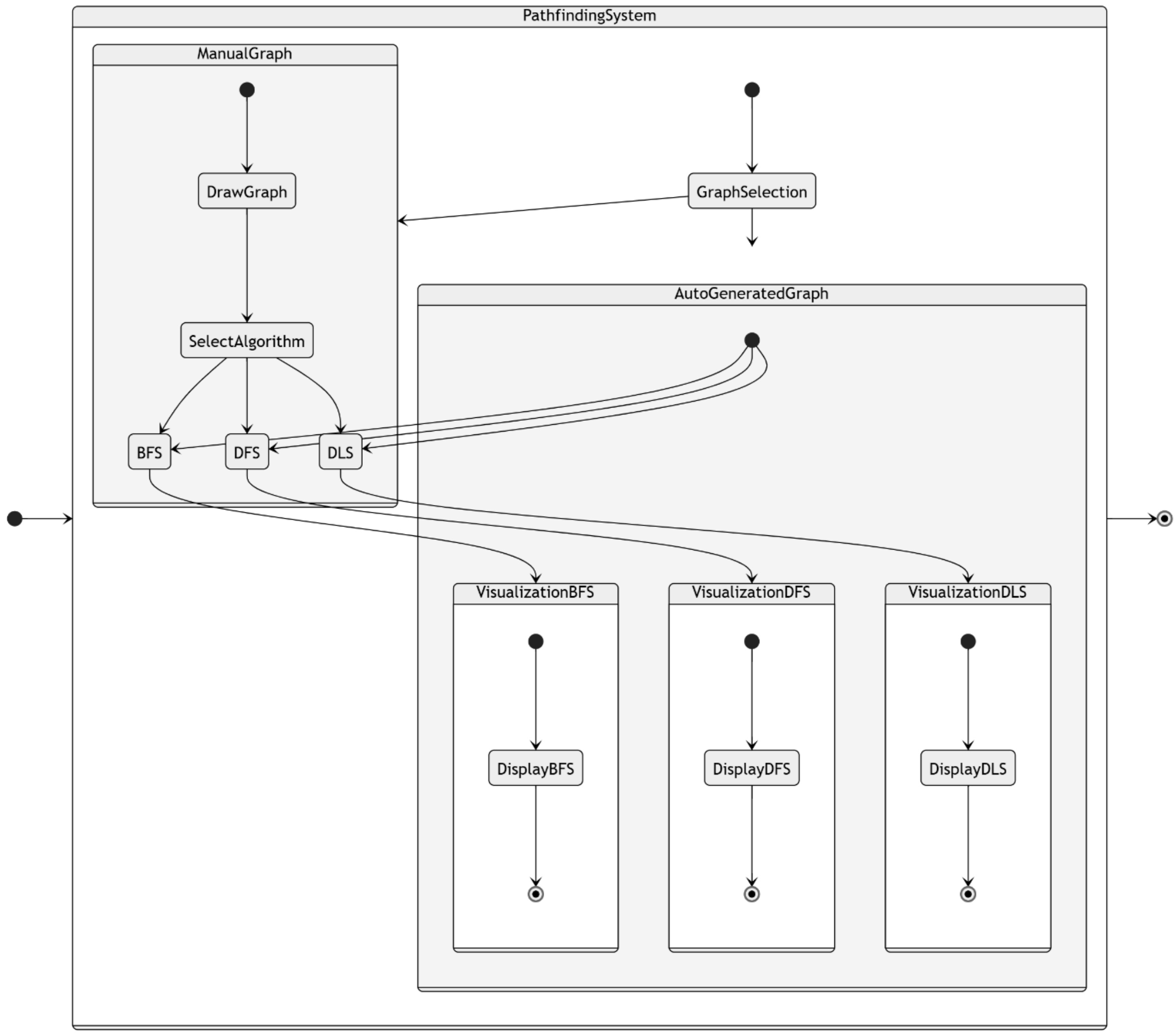

Fig 4.3 State Diagram

4.4 Gantt Chart

A Gantt Chart is a project management tool used to visualize the timeline, tasks, and dependencies of a project. It represents tasks as horizontal bars along a timeline, with the length of each bar corresponding to the duration of the task. The chart provides an overview of the project's schedule, allowing teams to track progress, identify critical dependencies, and ensure that milestones are met on time.

For the Pathfinding Algorithm Project, the Gantt Chart is instrumental in organizing and planning each phase, from requirements gathering to testing and deployment. Here's a breakdown of how it applies to this project:

1. Task Breakdown: The project is divided into manageable tasks, such as requirements gathering, algorithm implementation, UI design, and testing.
2. Timeline Allocation: Each task is assigned a specific start and end date based on its estimated duration.
3. Dependencies: Dependencies between tasks, such as testing requiring completed implementation, are clearly identified to avoid bottlenecks.
4. Progress Monitoring: The Gantt Chart allows regular updates on task completion, ensuring adherence to the schedule.
5. Resource Management: By visualizing tasks and timelines, the chart aids in resource allocation, such as assigning team members to specific tasks.

In this project, the Gantt Chart ensures a structured approach, enabling smooth transitions between phases like BFS, DFS, and DLS implementation, user interface design, and rigorous testing. It serves as a roadmap for achieving project milestones efficiently.

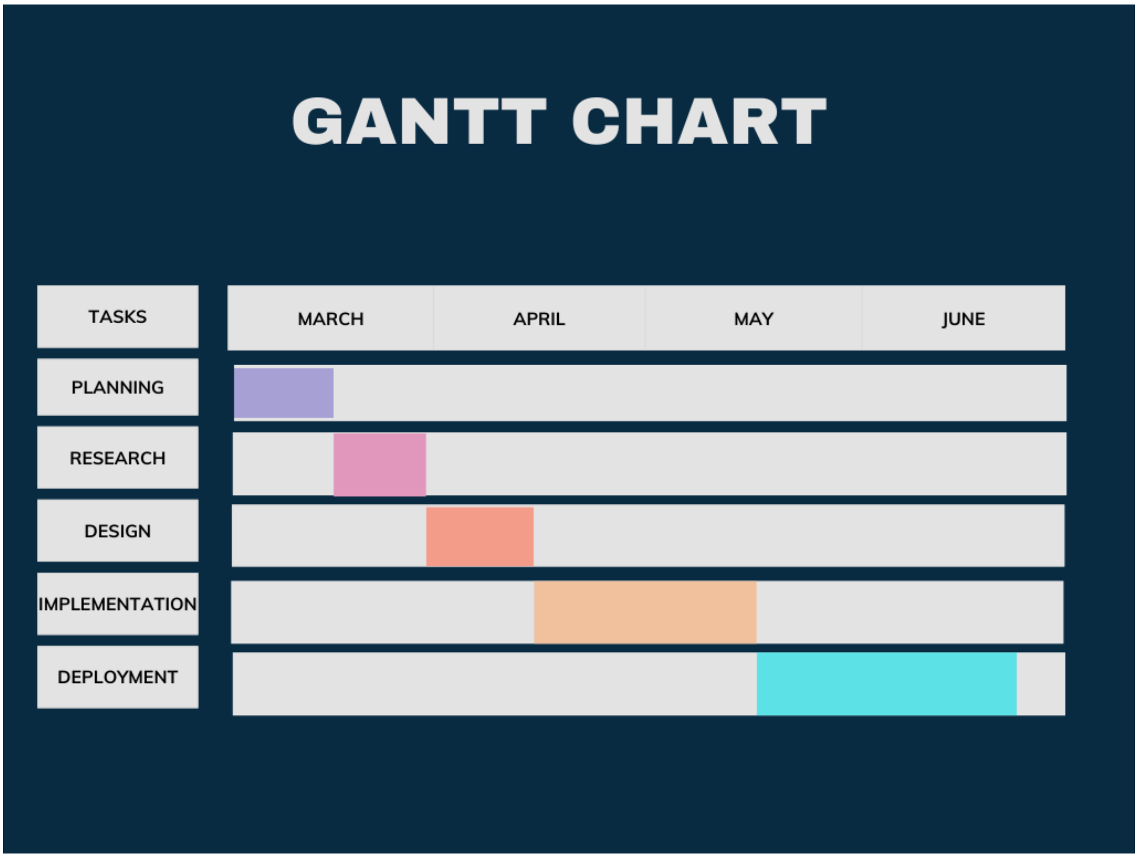

Fig 4.4 Gantt Chart

CHAPTER 5

IMPLEMENTATION AND CODING

5.1 Implementation Approach

The Pathfinder Pro project combines a diverse set of programming techniques, frameworks, and tools to deliver a seamless and interactive platform for visualizing graph algorithms. The implementation process focuses on integrating a user-friendly interface with powerful backend logic to process algorithms and manage data efficiently. The client-server architecture ensures smooth communication between the frontend and backend, allowing real-time updates and dynamic visualizations. The project leverages open-source technologies to maximize scalability, cost-effectiveness, and ease of use.

The frontend is built using HTML, CSS, and JavaScript to provide a visually appealing and responsive interface. It features interactive elements like graph creation, drag-and-drop functionality, and control panels for algorithm selection. The dynamic visualizations are powered by JavaScript, enabling step-by-step animations of BFS, DFS, and DLS traversals. The frontend also uses AJAX requests to communicate with the backend, ensuring a seamless experience without requiring page reloads. The design emphasizes accessibility and usability, catering to users of all technical skill levels.

On the backend Flask serves as the lightweight web framework, handling requests, routing, and algorithm execution. Python is the core programming language, chosen for its simplicity and extensive library support. NetworkX is utilized for creating and managing graph structures, while the `random` module facilitates the generation of randomized graphs for automated testing and demonstrations. The algorithms are implemented as reusable Python functions, making it easy to integrate additional functionalities in the future. By maintaining a modular code structure, the project ensures flexibility for scaling and debugging, laying a strong foundation for long-term development and enhancement.

5.2 Coding Details

Auto Generate Graph

```python
import tkinter as tk
from tkinter import ttk
import networkx as nx
import random

# Node colors
VISITED_COLOR = 'blue'
CURRENT_COLOR = 'green'
PATH_COLOR = 'red'


class PathfindingApp:
    def __init__(self, master):
        self.master = master
        self.master.title("Pathfinding Algorithms")
        self.master.geometry("1200x700")
        self.master.configure(bg="#D3D3D3")

        self.G = nx.Graph()
        self.node_positions = {}
        self.start_node = None
        self.end_node = None
        self.visited_nodes = set()
        self.current_algorithm = tk.StringVar(value="Select Algorithm")
        self.dls_limit = tk.IntVar(value=3)
        self.operations_queue = []  # Stores the sequence of operations for step-by-step navigation
        self.current_step = 0

        # GUI Layout
        self.style = ttk.Style()
        self.style.configure("TButton", font=("Arial", 12), padding=10)
        self.style.configure("TLabel", font=("Arial", 12))

        self.main_frame = ttk.Frame(self.master, style="TFrame")
        self.main_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

        self.canvas_frame = ttk.Frame(self.main_frame)
        self.canvas_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True, padx=20)

        self.sidebar_frame = tk.Frame(self.main_frame, bg="#D3D3D3")
        self.sidebar_frame.pack(side=tk.LEFT, fill=tk.Y, padx=20)

        self.algorithm_menu = tk.OptionMenu(
            self.sidebar_frame, self.current_algorithm, "BFS", "DFS", "DLS"
```

```python
    def generate_random_graph(self):
        """Generate a random graph."""
        self.G.clear()
        self.node_positions.clear()

        num_nodes = random.randint(5, 10)
        probability = random.uniform(0.2, 0.6)
        self.G = nx.gnp_random_graph(num_nodes, probability, seed=42)
        self.node_positions = nx.spring_layout(self.G, seed=42)

        self.scale_node_positions()
        self.draw_graph()

    def scale_node_positions(self):
        """Scale node positions to fit within the canvas size."""
        x_vals = [pos[0] for pos in self.node_positions.values()]
        y_vals = [pos[1] for pos in self.node_positions.values()]
        min_x, max_x = min(x_vals), max(x_vals)
        min_y, max_y = min(y_vals), max(y_vals)
        padding = 50
        canvas_width, canvas_height = 800, 600

        for node in self.node_positions:
            x, y = self.node_positions[node]
            scaled_x = padding + (x - min_x) / (max_x - min_x) * (canvas_width - 2 * padding)
            scaled_y = padding + (y - min_y) / (max_y - min_y) * (canvas_height - 2 * padding)
            self.node_positions[node] = (scaled_x, scaled_y)

    def select_node(self, event):
        """Select start and end nodes on the canvas."""
        x, y = event.x, event.y
        for node, pos in self.node_positions.items():
            node_x, node_y = int(pos[0]), int(pos[1])
            if abs(node_x - x) < 20 and abs(node_y - y) < 20:
                if not self.selecting_start and not self.selecting_end:
                    self.selecting_start = True
                    self.start_node = node
                    self.instruction_label.config(text="Select the end node.")
                    self.canvas.create_oval(node_x - 12, node_y - 12, node_x + 12, node_y + 12, fill="green", outline="black", tags="start_end")
                    return
                elif self.selecting_start and not self.selecting_end:
                    self.selecting_end = True
                    self.end_node = node
```

```python
                    self.end_node = node
                    self.instruction_label.config(text="Starting algorithm...")
                    self.canvas.create_oval(node_x - 12, node_y - 12, node_x + 12, node_y + 12, fill="red", outline="black", tags="start_end")
                    return

    def draw_graph(self):
        """Draw the graph on the canvas."""
        self.canvas.delete("all")
        for node, pos in self.node_positions.items():
            x, y = int(pos[0]), int(pos[1])
            self.canvas.create_oval(x - 10, y - 10, x + 10, y + 10, fill="gray", tags=f"node{node}")
            self.canvas.create_text(x, y - 15, text=str(node), font=("Arial", 14, "bold"), tags=f"node{node}")

        for edge in self.G.edges():
            node1, node2 = edge
            x1, y1 = self.node_positions[node1]
            x2, y2 = self.node_positions[node2]
            self.canvas.create_line(x1, y1, x2, y2)

    def start_algorithm(self):
        """Initialize the algorithm and prepare for step-by-step execution."""
        if self.start_node is None or self.end_node is None:
            self.instruction_label.config(text="Please select start and end nodes.")
            return

        self.visited_nodes.clear()
        self.operations_queue.clear()
        self.current_step = 0
        for row in self.treeview.get_children():
            self.treeview.delete(row)

        if self.current_algorithm.get() == "BFS":
            self.prepare_bfs()
        elif self.current_algorithm.get() == "DFS":
            self.prepare_dfs()
        elif self.current_algorithm.get() == "DLS":
            self.prepare_dls(self.dls_limit.get())

    def prepare_bfs(self):
```

```python
    def prepare_bfs(self):
        """Prepare BFS operations."""
        queue = [self.start_node]
        parent_map = {self.start_node: None}
        self.visited_nodes.add(self.start_node)
        self.operations_queue.append(("Start", self.start_node))

        while queue:
            current_node = queue.pop(0)
            self.operations_queue.append(("Dequeue", current_node))

            if current_node == self.end_node:
                self.highlight_path(parent_map)
                break

            for neighbor in self.G.neighbors(current_node):
                if neighbor not in self.visited_nodes:
                    queue.append(neighbor)
                    parent_map[neighbor] = current_node
                    self.visited_nodes.add(neighbor)
                    self.operations_queue.append(("Enqueue", neighbor))

    def prepare_dfs(self):
        """Prepare DFS operations."""
        stack = [self.start_node]
        parent_map = {self.start_node: None}
        self.visited_nodes.add(self.start_node)
        self.operations_queue.append(("Start", self.start_node))

        while stack:
            current_node = stack.pop()
            self.operations_queue.append(("Pop", current_node))

            if current_node == self.end_node:
                self.highlight_path(parent_map)
                break

            for neighbor in self.G.neighbors(current_node):
                if neighbor not in self.visited_nodes:
                    stack.append(neighbor)
                    parent_map[neighbor] = current_node
                    self.visited_nodes.add(neighbor)
                    self.operations_queue.append(("Push", neighbor))
```

```python
            self.current_step += 1

    def update_canvas(self, node, visited=False):
        """Update the canvas to reflect the current operation."""
        node_x, node_y = self.node_positions[node]

        if visited:
            self.canvas.create_oval(node_x - 10, node_y - 10, node_x + 10, node_y + 10, fill=VISITED_COLOR)
        else:
            self.canvas.create_oval(node_x - 10, node_y - 10, node_x + 10, node_y + 10, fill=CURRENT_COLOR)

        self.master.update()

    def update_table(self, operation, node):
        """Update the treeview table with the latest operation."""
        self.treeview.insert("", "end", values=(operation, node))

    def reset(self):
        """Reset the application to its initial state."""
        # Clear the canvas
        self.canvas.delete("all")

        # Reset graph state
        self.visited_nodes.clear()
        self.start_node = None
        self.end_node = None
        self.selecting_start = False
        self.selecting_end = False
        self.operations_queue.clear()
        self.current_step = 0

        # Reset instruction label
        self.instruction_label.config(text="Click on nodes to select start and end nodes.")

        # Clear the operation table
        self.treeview.delete(*self.treeview.get_children())

        # Generate a fresh random graph
        self.generate_random_graph()


# Run the application
root = tk.Tk()
app = PathfindingApp(root)
root.mainloop()
```

Manual Generate Graph

```python
import tkinter as tk
from tkinter import ttk
import networkx as nx

# Colors for nodes and interface
VISITED_COLOR = '#20B2AA'
CURRENT_COLOR = '#FFD700'
PATH_COLOR = '#FF00FF'
START_COLOR = '#FFA500'
GOAL_COLOR = '#9370DB'
BACKGROUND_COLOR = '#F8F9FA'
GRID_COLOR = '#E5E7EB'


class PathfindingApp:
    def __init__(self, master):
        self.master = master
        self.master.title("Pathfinding Algorithms")
        self.master.geometry("1200x700")
        self.master.config(bg=BACKGROUND_COLOR)

        # Initialize the graph
        self.G = nx.Graph()
        self.node_positions = {}
        self.start_node = None
        self.end_node = None
        self.selecting_edge = False
        self.selected_node1 = None
        self.visited_nodes = set()
        self.path_nodes = set()
        self.current_algorithm = tk.StringVar(value="Select Algorithm")
        self.operations_queue = []  # Queue to hold operations for next_step
        self.current_step = 0  # To keep track of the current step in the queue

        # Main layout frames
        self.main_frame = ttk.Frame(self.master)
        self.main_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

        self.canvas_frame = ttk.Frame(self.main_frame)
        self.canvas_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True, padx=20)

        self.sidebar_frame = ttk.Frame(self.main_frame, padding=(15, 10))
        self.sidebar_frame.pack(side=tk.LEFT, fill=tk.Y, padx=20, pady=20)

        # Sidebar elements
        title = ttk.Label(self.sidebar_frame, text="Pathfinding Visualizer", font=("Helvetica", 16, "bold"))
        title.pack(pady=(0, 20))

        algo_label = ttk.Label(self.sidebar_frame, text="Select Algorithm:", font=("Helvetica", 12))
        algo_label.pack()
        self.algorithm_menu = tk.OptionMenu(self.sidebar_frame, self.current_algorithm, "BFS", "DFS")
        self.algorithm_menu.config(width=15)
        self.algorithm_menu.pack(pady=(5, 15))

```

```python
    def __init__(self, master):
        self.algorithm_menu = tk.OptionMenu(
        )
        self.algorithm_menu.pack(pady=15)

        self.dls_limit_label = ttk.Label(self.sidebar_frame, text="Set Depth Limit for DLS:")
        self.dls_limit_label.pack(pady=5)
        self.dls_limit_input = ttk.Entry(self.sidebar_frame, textvariable=self.dls_limit, width=5)
        self.dls_limit_input.pack(pady=5)

        self.instruction_label = ttk.Label(self.sidebar_frame, text="Click on nodes to select start and end nodes.")
        self.instruction_label.pack(pady=10)

        self.start_button = ttk.Button(self.sidebar_frame, text="Start Algorithm", command=self.start_algorithm)
        self.start_button.pack(pady=10)

        self.reset_button = ttk.Button(self.sidebar_frame, text="Reset", command=self.reset)
        self.reset_button.pack(pady=10)

        self.next_button = ttk.Button(self.sidebar_frame, text="Next Step", command=self.next_step)
        self.next_button.pack(pady=15)

        self.graph_selection_button = ttk.Button(
            self.sidebar_frame, text="Generate Random Graph", command=self.generate_random_graph
        )
        self.graph_selection_button.pack(pady=15)

        self.table_frame = ttk.Frame(self.sidebar_frame)
        self.table_frame.pack(pady=10)
        self.treeview = ttk.Treeview(self.table_frame, columns=("Operation", "Node"), show="headings", height=10)
        self.treeview.heading("Operation", text="Operation")
        self.treeview.heading("Node", text="Node")
        self.treeview.pack()

        self.canvas = tk.Canvas(self.canvas_frame, width=1000, height=800, bg="white")
        self.canvas.pack()

        self.selecting_start = False
        self.selecting_end = False
        self.canvas.bind("<Button-1>", self.select_node)

    def generate_random_graph(self):
        """Generate a random graph."""
```

```python
        self.start_button = ttk.Button(self.sidebar_frame, text="Start Algorithm", command=self.start_algorithm)
        self.start_button.pack(pady=10)

        self.reset_button = ttk.Button(self.sidebar_frame, text="Reset", command=self.reset)
        self.reset_button.pack(pady=10)

        self.next_step_button = ttk.Button(self.sidebar_frame, text="Next Step", command=self.next_step, state=tk.DISABLED)
        self.next_step_button.pack(pady=10)

        self.toggle_edge_button = ttk.Button(self.sidebar_frame, text="Toggle Edge Creation Mode", command=self.toggle_edge_mode)
        self.toggle_edge_button.pack(pady=10)

        self.select_start_button = ttk.Button(self.sidebar_frame, text="Select Start Node", command=self.enable_select_start)
        self.select_start_button.pack(pady=5)

        self.select_goal_button = ttk.Button(self.sidebar_frame, text="Select Goal Node", command=self.enable_select_goal)
        self.select_goal_button.pack(pady=5)

        self.table_frame = ttk.LabelFrame(self.sidebar_frame, text="Stack/Queue Operations", padding=(10, 5))
        self.table_frame.pack(pady=(15, 10), fill=tk.X)
        self.treeview = ttk.Treeview(self.table_frame, columns=("Operation", "Node"), show="headings", height=8)
        self.treeview.heading("Operation", text="Operation")
        self.treeview.heading("Node", text="Node")
        self.treeview.pack(fill=tk.X)

        self.instruction_label = ttk.Label(self.sidebar_frame, text="Click on the canvas to add nodes.", wraplength=180)
        self.instruction_label.pack(pady=(15, 10))

        self.canvas = tk.Canvas(self.canvas_frame, width=800, height=600, bg="white")
        self.canvas.pack(fill=tk.BOTH, expand=True)
        self.draw_grid()

        self.canvas.bind("<Button-1>", self.select_node)

        self.selecting_start = False
        self.selecting_goal = False

    def draw_grid(self):
        for i in range(0, 800, 20):
            self.canvas.create_line([(i, 0), (i, 600)], fill=GRID_COLOR, tags='grid_line')
        for i in range(0, 600, 20):
            self.canvas.create_line([(0, i), (800, i)], fill=GRID_COLOR, tags='grid_line')

    def enable_select_start(self):
        self.selecting_start = True
        self.selecting_goal = False
        self.instruction_label.config(text="Click on a node to set it as the start node.")

    def enable_select_goal(self):
        self.selecting_goal = True
        self.selecting_start = False
```

```python
            self.instruction_label.config(text="Click on a node to set it as the goal node.")

    def select_node(self, event):
        x, y = event.x, event.y
        if self.selecting_edge:
            self.handle_edge_creation(x, y)
            return

        if self.selecting_start or self.selecting_goal:
            selected_node = self.get_node_at_position(x, y)
            if selected_node:
                if self.selecting_start:
                    self.start_node = selected_node
                    self.update_canvas(selected_node, color=START_COLOR)
                    self.instruction_label.config(text="Start node selected.")
                    self.selecting_start = False
                elif self.selecting_goal:
                    self.end_node = selected_node
                    self.update_canvas(selected_node, color=GOAL_COLOR)
                    self.instruction_label.config(text="Goal node selected.")
                    self.selecting_goal = False
            return

        node_name = f"Node{len(self.node_positions) + 1}"
        self.node_positions[node_name] = (x, y)
        self.draw_graph()

    def handle_edge_creation(self, x, y):
        if self.selected_node1 is None:
            self.selected_node1 = self.get_node_at_position(x, y)
            if self.selected_node1:
                self.instruction_label.config(text="Click on another node to create the edge.")
        else:
            selected_node2 = self.get_node_at_position(x, y)
            if selected_node2 and selected_node2 != self.selected_node1:
                self.G.add_edge(self.selected_node1, selected_node2)
                self.canvas.create_line(
                    self.node_positions[self.selected_node1][0], self.node_positions[self.selected_node1][1],
                    self.node_positions[selected_node2][0], self.node_positions[selected_node2][1],
                    fill="black", width=2
                )
                self.selected_node1 = None
                self.selecting_edge = False
                self.instruction_label.config(text="Edge created.")

    def get_node_at_position(self, x, y):
        for node, pos in self.node_positions.items():
            node_x, node_y = pos
            if abs(node_x - x) < 20 and abs(node_y - y) < 20:
                return node
        return None

```

```python

    def draw_graph(self):
        self.canvas.delete("all")
        self.draw_grid()
        for node, pos in self.node_positions.items():
            x, y = pos
            color = "gray"
            if node == self.start_node:
                color = START_COLOR
            elif node == self.end_node:
                color = GOAL_COLOR
            elif node in self.visited_nodes:
                color = VISITED_COLOR
            elif node in self.path_nodes:
                color = PATH_COLOR
            self.canvas.create_oval(x-10, y-10, x+10, y+10, fill=color, tags=f"node_{node}")
            self.canvas.create_text(x, y-15, text=node, font=("Arial", 14, "bold"), tags=f"node_{node}")

        for edge in self.G.edges:
            node1, node2 = edge
            x1, y1 = self.node_positions[node1]
            x2, y2 = self.node_positions[node2]
            self.canvas.create_line(x1, y1, x2, y2, fill="black", width=2)

    def start_algorithm(self):
        self.visited_nodes.clear()
        self.path_nodes.clear()
        self.current_step = 0
        self.operations_queue.clear()
        self.treeview.delete(*self.treeview.get_children())

        self.start_button.config(state=tk.DISABLED)
        self.reset_button.config(state=tk.DISABLED)
        self.next_step_button.config(state=tk.NORMAL)

        algorithm = self.current_algorithm.get()

        if algorithm == "BFS":
            self.bfs(self.start_node)
        elif algorithm == "DFS":
            self.dfs(self.start_node)

    def bfs(self, start_node):
        queue = [start_node]
        visited = set([start_node])
        self.operations_queue.append(("Start", start_node))
        while queue:
            node = queue.pop(0)
            self.visited_nodes.add(node)
            self.update_canvas(node, color=VISITED_COLOR)
            self.operations_queue.append(("Dequeue", node))
            for neighbor in self.G.neighbors(node):
```

```python
                    self.visited_nodes.add(neighbor)
                    self.operations_queue.append(("Push", neighbor))

    def prepare_dls(self, depth_limit):
        """Prepare DLS operations."""
        stack = [(self.start_node, 0)]
        parent_map = {self.start_node: None}
        self.visited_nodes.add(self.start_node)
        self.operations_queue.append(("Start", self.start_node))

        while stack:
            current_node, depth = stack.pop()
            self.operations_queue.append(("Pop", current_node))

            if current_node == self.end_node:
                self.highlight_path(parent_map)
                break

            if depth < depth_limit:
                for neighbor in self.G.neighbors(current_node):
                    if neighbor not in self.visited_nodes:
                        stack.append((neighbor, depth + 1))
                        parent_map[neighbor] = current_node
                        self.visited_nodes.add(neighbor)
                        self.operations_queue.append(("Push", neighbor))

    def highlight_path(self, parent_map):
        """Highlight the path from start to end."""
        current = self.end_node
        while current is not None:
            parent = parent_map.get(current)
            if parent is not None:
                x1, y1 = self.node_positions[parent]
                x2, y2 = self.node_positions[current]
                self.canvas.create_line(x1, y1, x2, y2, fill=PATH_COLOR, width=3)
            current = parent

    def next_step(self):
        """Execute the next step in the operations queue."""
        if self.current_step >= len(self.operations_queue):
            self.instruction_label.config(text="Algorithm Complete.")
            return

        operation, node = self.operations_queue[self.current_step]
        self.update_table(operation, node)
        if operation in ("Enqueue", "Dequeue", "Push", "Pop"):
            self.update_canvas(node, visited=(operation == "Dequeue"))

        self.current_step += 1
```

```python
        self.operations_queue.clear()
        self.treeview.delete(*self.treeview.get_children())
        self.selecting_edge = False
        self.selected_node1 = None
        self.selecting_start = False
        self.selecting_goal = False
        self.start_button.config(state=tk.NORMAL)
        self.reset_button.config(state=tk.NORMAL)
        self.next_step_button.config(state=tk.DISABLED)
        self.instruction_label.config(text="Click on the canvas to add nodes.")
        self.draw_graph()

    def update_canvas(self, node, color):
        x, y = self.node_positions[node]
        self.canvas.create_oval(x-10, y-10, x+10, y+10, fill=color, tags=f"node_{node}")

    def toggle_edge_mode(self):
        self.selecting_edge = not self.selecting_edge
        self.instruction_label.config(text="Edge creation mode activated. Click two nodes to create an edge."
                                      if self.selecting_edge else "Edge creation mode deactivated.")

# Run the application
root = tk.Tk()
app = PathfindingApp(root)
root.mainloop()
```

CHAPTER 6

SOFTWARE TESTING

6.1 Unit Testing

Unit testing focuses on testing individual components or functions of the application in isolation. For this project, key functions like graph generation, BFS, DFS, and DLS algorithms, as well as UI components such as canvas rendering and table updates, are tested independently. The goal is to verify that each module behaves as expected.

- Example: Testing whether the BFS algorithm correctly identifies the shortest path in a simple graph.

Key Areas:

- Node and edge creation
- Graph traversal logic
- Node and edge visualization

6.2 Integration Testing

Integration testing examines how different modules of the application work together. In this project, integration tests focus on interactions between the graph generation, algorithm execution, UI components, and canvas rendering.

- Example: Verifying that a generated graph is properly visualized on the canvas and that the selected algorithm operates correctly on it.

Key Areas:

- Interaction between graph logic and UI
- Synchronization of stack/queue updates with visual representation
- End-to-end testing of user workflows (e.g., selecting nodes and running algorithms)

6.3 Test Cases

Table 6.1 : Test Case Table for Pathfinding Algorithm Implementation

Test Case ID	Test Scenario	Test Steps	Test Data	Expected Result	Actual Result	Status
TC001	Verify Graph Generation Functionality	1. Launch the application. 2. Click on "Generate	N/A	A random graph is generated and	Graph generated successfully.	Passed

		Random Graph" button.		displayed on the canvas.		
TC002	Verify Start Node Selection	1. Generate a graph. 2. Click on a node to set it as the start node.	Node position on canvas.	Selected node changes color to indicate it is the start node.	Start node selected and highlighted.	Passed
TC003	Verify End Node Selection	1. Generate a graph. 2. Click on a node to set it as the end node.	Node position on canvas.	Selected node changes color to indicate it is the end node.	End node selection Passed.	Passed
TC004	BFS Algorithm Execution	1. Generate a graph. 2. Select start and end nodes. 3. Run the BFS algorithm.	Graph with connected nodes.	BFS identifies the shortest path and displays it.	BFS executed successfully.	Passed
TC005	DFS Algorithm Execution	1. Generate a graph. 2. Select start and end nodes. 3. Run the DFS algorithm.	Graph with connected nodes.	DFS traverses and highlights the discovered path.	DFS execution runs expectedly.	Passed

TC006	DLS Algorithm Execution	1. Generate a graph. 2. Select start and end nodes. 3. Set a depth limit. 4. Run DLS algorithm.	Graph with connected nodes; Depth limit.	DLS traverses the graph up to the specified depth limit.	DLS executes to respect the depth limit.	Passed
TC007	Reset Functionality	1. Generate a graph. 2. Select start and end nodes. 3. Click the "Reset" button.	N/A	All selections and visualizations are cleared.	Reset functionality worked correctly.	Passed
TC008	UI Responsiveness	1. Open the application on different screen resolutions.	Screen resolutions (e.g., 1080p, 720p)	UI elements are aligned and function properly across screens.	UI elements misaligned on smaller screens.	Failed
TC009	Invalid Node Selection	1. Generate a graph. 2. Click outside any node area on the canvas.	Empty click on canvas.	No action occurs; no node is selected.	Invalid selections were ignored.	Passed
TC010	Verify Path Reconstruction	1. Execute BFS, DFS, or DLS with valid start and end nodes.	Graph with a path between nodes.	The reconstructed path is highlighted and visualized.	Path reconstruction failed (visual issue).	Failed

TC011	Depth Limit Handling (DLS)	1. Generate a graph. 2. Set depth limit greater than graph depth. 3. Execute DLS.	Depth limit exceeding graph depth.	Algorithm terminates without finding a path, if applicable.	Algorithm terminates gracefully.	Passed

CHAPTER 7

RESULTS AND DISCUSSION

7.1 Running Project Screen Shots

Frontpage

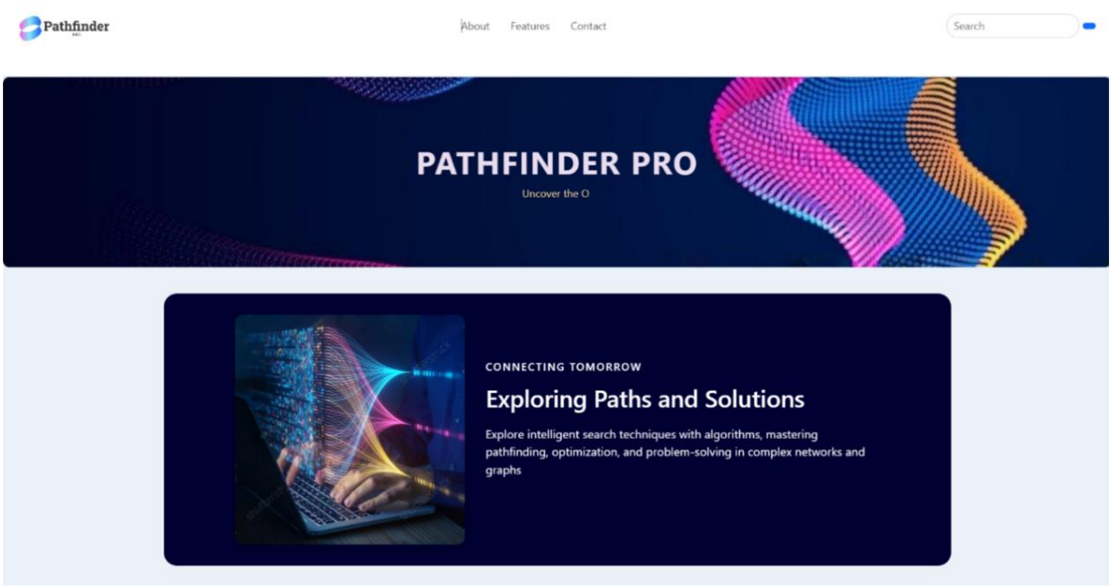

Fig 7.1 : Home Page

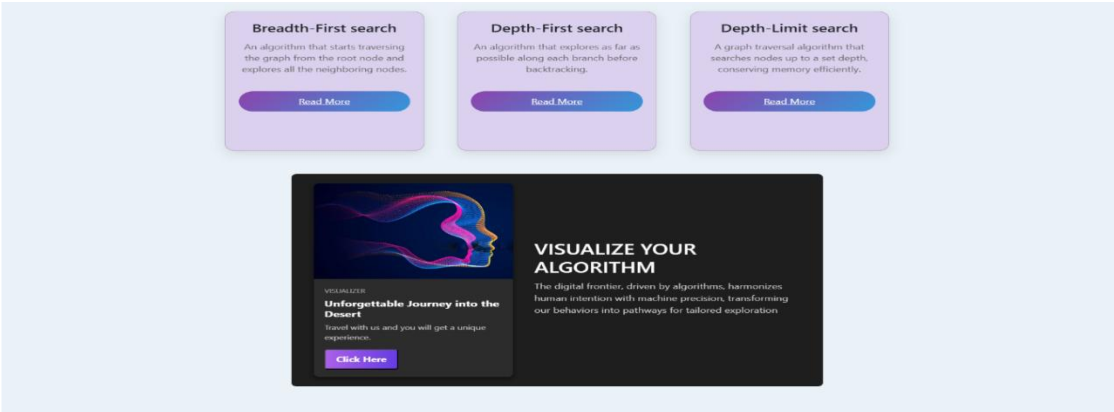

Fig 7.2 : Theory of Algorithms

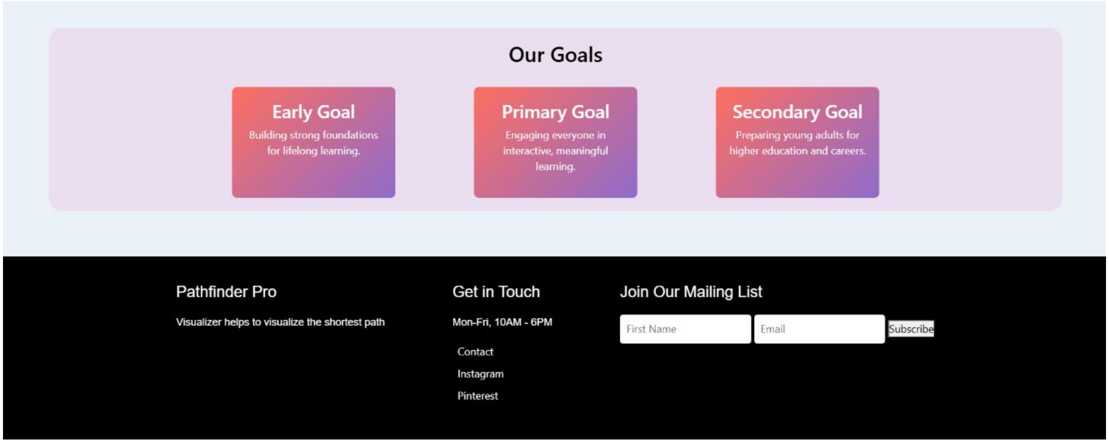

Fig 7.3 : Contact Details

Mode Selection

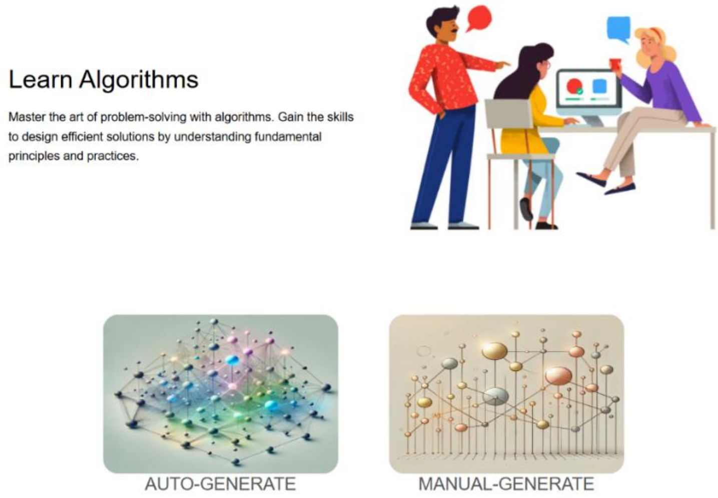

Fig 7.4 : Mode Selection Page

Auto Generate Graph

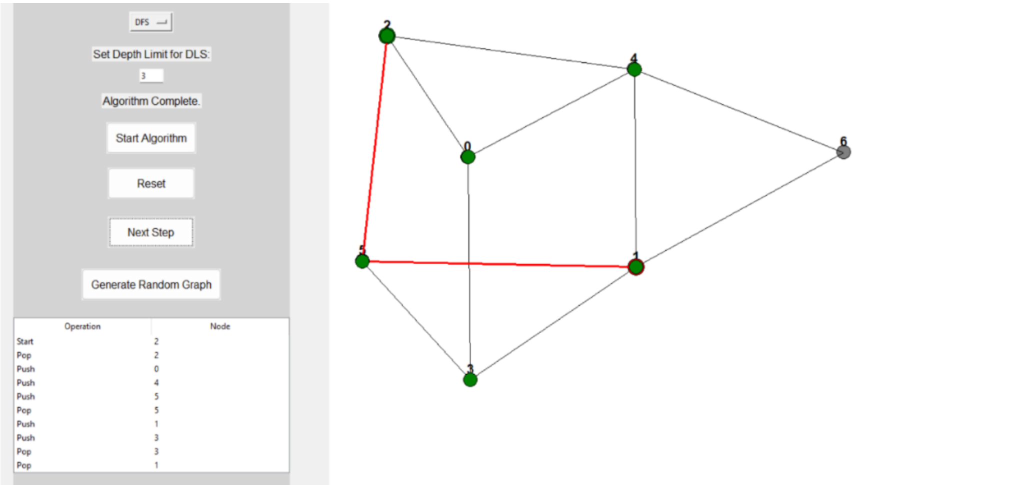

Fig 7.5 : Auto Generate Graph selection

Manual Graph

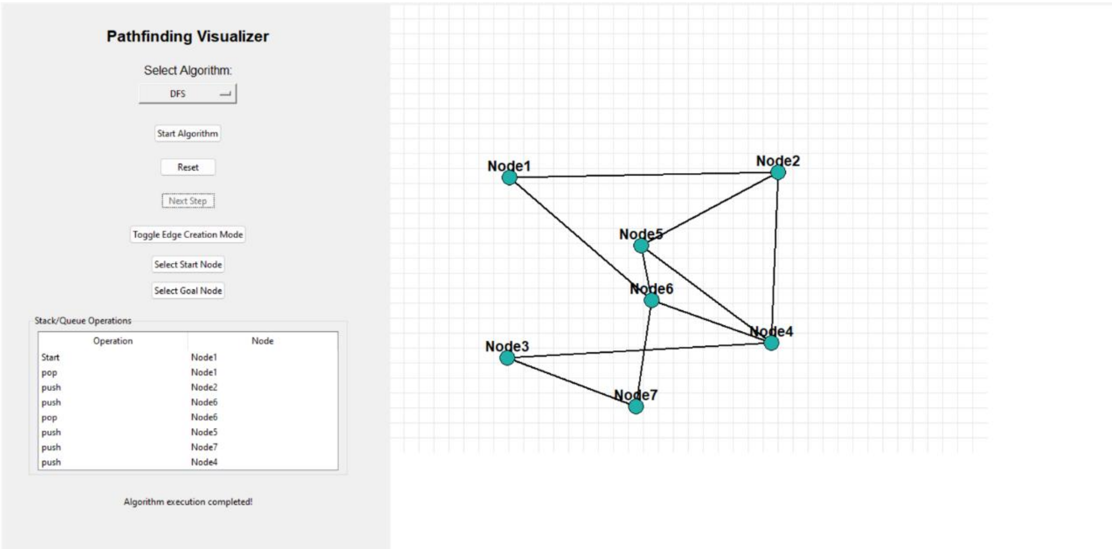

Fig 7.6 : Manual Graph Selection

7.2 User Documentation

7.2.1Services

The Pathfinder Pro project offers a range of services aimed at facilitating the understanding and application of graph algorithms. These services are designed to provide an interactive, user-friendly, and educational experience for users, ensuring the platform caters to both beginners and advanced learners. Below are the key services provided by the project:

7.2.1.1 Custom Graph Creation

Users can create their own graphs by adding nodes, edges, and assigning weights. This feature allows for full customization, enabling learners to experiment with various graph structures and observe how different algorithms behave in diverse scenarios.

7.2.1.2 Auto-Generated Graphs

For quick demonstrations or testing, the platform includes a feature to generate random graphs automatically. These graphs simulate real-world network structures, making it easier to understand the practical applications of graph algorithms.

7.2.1.3 Algorithm Visualization

The platform provides real-time visualization of graph traversal algorithms, including BFS, DFS, and DLS. Users can watch as nodes are visited, edges are traversed, and paths are computed step-by-step, offering an intuitive understanding of the algorithms' inner workings.

CHAPTER 8

CONCLUSION

8.1 Conclusion

The project achieved its objectives by delivering a robust and flexible solution that:

Visualizes Algorithms: Real-time, step-by-step animations of graph traversal enhance the learning process by making abstract concepts tangible and engaging.

Enables Customization: Users can create and manipulate their own graphs, experimenting with different scenarios to deepen their understanding of algorithm behaviors.

Integrates Advanced Technologies: The combination of Flask, NetworkX, and frontend technologies like HTML, CSS, and JavaScript ensures a scalable and efficient platform.

Supports Educational Goals: The platform serves as an educational resource, offering insights into graph theory concepts and their real-world applications in mapping, navigation, and network analysis.

8.2 Limitations and Challenges

While the project meets its goals, certain limitations exist:

The application's performance may decline with extremely large graphs due to computational overhead.

Advanced features such as weighted graphs and alternative algorithms (e.g., Dijkstra or A*) could be added to expand functionality.

Cross-browser compatibility issues may arise with older browsers.

FUTURE SCOPE OF THE PROJECT

The Pathfinder Pro project offers a strong foundation for future development, with significant potential to evolve into a comprehensive educational and analytical tool. One of the primary areas for growth is the integration of additional algorithms, such as Dijkstra’s and Bellman-Ford, to handle weighted graphs and advanced shortest-path calculations. These enhancements will expand the platform’s applicability to real-world problems, including transportation networks, communication routing, and logistics optimization. Furthermore, the introduction of heuristic-based algorithms like A* could bring intelligent pathfinding capabilities, enhancing the platform's utility in dynamic and complex scenarios.

In addition to algorithmic expansion, the project can explore advanced technologies such as AI and machine learning for analyzing large-scale graphs and detecting patterns in real-world networks like social media or transportation systems. Another important avenue is mobile compatibility, which would extend the platform’s accessibility to a broader audience, enabling users to interact with the tool on smartphones and tablets. The inclusion of multi-user collaboration features, such as shared graph creation and analysis, would further enhance its educational value by fostering teamwork and real-time collaboration in classroom and research environments. These developments promise to transform Pathfinder Pro into a scalable and versatile platform with broad applicability.

REFERENCES

1. Ahuja, R. K., Magnanti, T. L., & Orlin, J. B. (1993). Network Flows: Theory, Algorithms, and Applications. Prentice-Hall.
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). The MIT Press.
3. Harel, D., & Tarjan, R. E. (1984). Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2), 338-355.
4. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.
5. Pohl, I. (1970). Bi-directional Search. *SIGACT News*, 1(4), 6-7.
6. Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2), 146-160.
7. Knuth, D. E. (1975). The Art of Computer Programming, Volume 1: Fundamental Algorithms (2nd ed.). Addison-Wesley.
8. Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1, 269-271.
9. BFS Algorithm (2020). *GeeksforGeeks*. Retrieved from https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
10. DFS Algorithm (2020). *GeeksforGeeks*. Retrieved from https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
11. Floyd, R. W. (1962). Algorithm 97: Shortest Path. *Communications of the ACM*, 5(6), 345.
12. Benedict, M., & Schensted, T. (1982). A Survey of Pathfinding Algorithms. *Journal of Algorithms*, 3(1), 17-38.
13. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100-107.
14. Matuszek, D. (2000). Graph Search Algorithms and Their Applications in AI. *Journal of Artificial Intelligence*, 28(4), 325-339.
15. Garey, M. R., & Johnson, D. S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. *Freeman and Company*.