CHANDIGARH UNIVERSITY
Discover. Learn. Empower.

NAAC GRADE A+
Accredited University

UNIVERSITY INSTITUTE of COMPUTING
Asia's Fastest Growing University

Mini Project

ON

Data Structure Visualizer

Submitted By:

Name: Atul Sinha

Balram Kumar

UID: 24MCA20010

24MCA20020

Course: MCA

Section: 24MCA-1 A

Subject: Design & Analysis of Algorithm

Semester: 2nd

Submitted To:

Ms. **Kawaljit Kaur**

TABLE OF CONTENTS

Topic	Page No.
ABSTRACT	3
CHAPTER-1: Introduction	3
1.i. Objective	3
1.ii. System Specifications	3
CHAPTER-2: Literature Review	4
CHAPTER-3: Coding	5
CHAPTER-4: Outputs	16
CHAPTER-5: Implementation Details	21
CHAPTER-6: System Study	22
CHAPTER-7: Technical Feasibility	23
CHAPTER-8: Conclusions	24
CHAPTER-9: References	24

Abstract

This Python program is an interactive graphical application for visualizing sorting and graph traversal algorithms, developed using the `Tkinter` library. The application offers an intuitive interface where users can select from sorting algorithms like Bubble Sort, Selection Sort, and Quick Sort, as well as graph traversal algorithms such as Breadth-First Search (BFS), Dijkstra's Algorithm, and Prim's Algorithm. Each selected algorithm is visualized in real-time on a canvas, with sorting algorithms represented by color-coded bars and graph traversal algorithms by grids or graph nodes and edges. The app's threading mechanism ensures smooth GUI performance by running each visualization on a separate thread. Additionally, a speed control slider enables users to adjust the animation speed, enhancing interactive learning. This application provides an engaging way to observe algorithmic processes, allowing users to gain insights into the mechanics of each algorithm through visual feedback and active highlighting of elements during execution.

1. Introduction

i. **Objective:** The primary objective of this application is to provide an engaging and educational tool for visualizing fundamental algorithms, helping users, particularly students and beginners in computer science, to understand complex algorithmic processes through interactive animations. By allowing users to select specific algorithms and observe their step-by-step execution, the application aims to make learning abstract concepts, such as sorting and graph traversal, more accessible and intuitive. The application's secondary objective is to maintain a responsive and user-friendly interface, incorporating a dynamic speed control feature and real-time updates. This encourages experimentation with different algorithms, thereby fostering a deeper, hands-on learning experience. Through this tool, the application seeks to bridge the gap between theoretical understanding and practical visualization, supporting users in grasping core algorithmic concepts more effectively.

ii. **System Specification:** The system specifications for running this algorithm visualization application are as follows:
   - **Programming Language:** Python 3.7 or above is required, as the application is built entirely in Python.

- **Libraries**:
    - **Tkinter**: For GUI development, generally this library comes included with standard Python installation.
    - **Threading** and **time**: Used for managing concurrent visualizations and setting animation speeds.
    - **Heapq**: Required for Dijkstra's and Prim's algorithms that is for priority queue operations.
    - **Random**: To generate sample data arrays and graph weights.
- **Hardware Requirements**:
    - **Processor**: Dual-core processor or better, as the visualization updates can be CPU-intensive during animation.
    - **Memory:** 4GB RAM minimum to handle concurrent thread execution smoothly.
    - **Graphics**: Basic integrated graphics are sufficient since the Tkinter library renders graphics on the CPU.
- **Operating System**: Compatible with Windows, macOS, and Linux as long as Python 3 is supported.
- **Display**: A minimum screen resolution of 1280x720 is recommended to ensure the UI elements and canvas are displayed comfortably.

2. Literature Review

A literature review of algorithm visualization highlights the educational and cognitive benefits of interactive visual aids in understanding complex computational processes. It is recognized that visualizing algorithms helps bridge the gap between abstract concepts and practical understanding, especially for sorting and graph traversal algorithms, which form a foundational part of computer science education.

a. **Educational Impact of Algorithm Visualization**: Several studies emphasize the value of algorithm visualization tools in enhancing students' comprehension and engagement. For example, studies by Hundhausen et al. (2002) and Shaffer et al. (2010) illustrate that students who engage with visual representations of algorithms demonstrate improved understanding and retention compared to those who learn solely from textual explanations. By providing real-time feedback and animations, algorithm visualizers help

clarify algorithmic steps, enabling students to observe data transformations and state changes that are otherwise challenging to conceptualize.

b. **Types of Visualization**: Algorithm visualization tools often focus on sorting and graph traversal, as these algorithms have dynamic behaviors suited to visual representation. Sorting algorithms, such as Bubble Sort and Quick Sort, are commonly represented by color-coded bars whose positions and colors change to indicate sorting steps, making the algorithms' progress visually intuitive. Graph traversal algorithms, including Breadth-First Search (BFS) and Dijkstra's Algorithm, utilize node and edge visualizations to depict the traversal path and shortest path calculations, thereby reinforcing spatial understanding of graph structures. Early work by Stasko (1997) paved the way for these visualizations, demonstrating how color changes, node highlights, and edge connections.
c. **Cognitive Load and Visualization Techniques**: A significant aspect of visualization design is managing cognitive load. Studies indicate that poor or overly complex visualizations can hinder learning by overwhelming users. Efficient visualizations avoid unnecessary detail and prioritize key algorithmic elements, such as comparison points in sorting or the shortest path in graph traversal, to reduce cognitive burden.
d. **Application of Algorithm Visualization in Education**: Algorithm visualization tools have been implemented in academic curricula, often through web-based platforms or desktop applications, providing students with easy access to interactive learning aids. Integrating visualization with traditional teaching methods has shown to significantly improve conceptual understanding and student performance. Several platforms, like the Python-based Tkinter library, offer a foundation for building interactive visualizations, enabling students and educators to create custom applications suited to specific educational needs.

In summary, literature supports that algorithm visualization tools, when designed with interactivity and cognitive simplicity in mind, provide substantial benefits in computer science education. They serve not only as learning aids but also as frameworks for self-paced, interactive exploration, fostering a deeper and more intuitive understanding of complex algorithmic processes.

3. Coding

```
import tkinter as tk
```

```
from tkinter import ttk
import time
import random
import threading
import heapq



def threaded(func):
    def wrapper(*args, **kwargs):
        thread = threading.Thread(target=func, args=args, kwargs=kwargs)
        thread.start()
    return wrapper

class AlgorithmVisualizerApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Algorithm Visualizer")
        self.root.geometry("700x800")
        self.title_label = tk.Label(
            root, text="Select Algorithm to Visualize", font=("Arial", 16)
        )
        self.title_label.pack(pady=10)
        self.algo_type_label = tk.Label(
            root, text="Choose Algorithm Type:", font=("Arial", 12)
        )
        self.algo_type_label.pack()
        self.algo_type = ttk.Combobox(root, values=["Sorting", "Graph
Traversal"])
        self.algo_type.pack(pady=5)
        self.algo_type.bind("<<ComboboxSelected>>", self.update_algorithm_list)
        self.algo_label = tk.Label(
            root, text="Choose Specific Algorithm:", font=("Arial", 12)
        )
        self.algo_label.pack()
        self.algorithm = ttk.Combobox(root, values=[])
        self.algorithm.pack(pady=5)
        self.run_button = tk.Button(
            root, text="Run Visualization", command=self.run_visualization
        )
```

```
                                self.run_button.pack(pady=20)
        self.speed_scale = tk.Scale(
            root,
            from_=0.1,
            to=2.0,


            resolution=0.1,
          orient=tk.HORIZONTAL,
          label="Visualization Speed",
        )
        self.speed_scale.set(1.0)
        self.speed_scale.pack(pady=10)
        self.canvas = tk.Canvas(root, width=600, height=350, bg="white")
        self.canvas.pack(pady=10)
        self.status_label = tk.Label(root, text="", font=("Arial", 10), fg="blue")
        self.status_label.pack()

    def update_algorithm_list(self, event):
        self.algorithm.set("")
        algo_type = self.algo_type.get()
        if algo_type == "Sorting":
            self.algorithm["values"] = [
                "Bubble Sort",
                "Selection Sort",
                "Quick Sort",
            ]
        elif algo_type == "Graph Traversal":
            self.algorithm["values"] = [
                "Breadth-First Search (BFS)",
                "Dijkstra's Algorithm",
                "Prim's Algorithm",
            ]
        else:
            self.algorithm["values"] = []

    @threaded
    def run_visualization(self):
        algo_type = self.algo_type.get()
```

```
                algo_name =
self.algorithm.get()

        if algo_type == "Sorting":
            if algo_name == "Bubble Sort":
                self.visualize_bubble_sort()
            elif algo_name == "Selection Sort":


                self.visualize_selection_sort()
            elif algo_name == "Quick Sort":
                self.visualize_quick_sort()
            else:
                self.status_label.config(
                    text="Please select a valid sorting algorithm."
                )
        elif algo_type == "Graph Traversal":
            if algo_name == "Breadth-First Search (BFS)":
                self.visualize_bfs()
            elif algo_name == "Dijkstra's Algorithm":
                self.visualize_dijkstra()
            elif algo_name == "Prim's Algorithm":
                self.visualize_prim()
            else:
                self.status_label.config(
                    text="Please select a valid graph traversal algorithm."
                )
        else:
            self.status_label.config(text="Please select a valid algorithm type.")

    def visualize_bubble_sort(self):
        self.status_label.config(text="Running Bubble Sort Visualization...")
        array = [random.randint(10, 100) for _ in range(20)]
        self.run_sort_algorithm(array, self.bubble_sort)

    def bubble_sort(self, array, draw_array):
        n = len(array)
        for i in range(n):
            for j in range(0, n - i - 1):
```

```
if array[j] > array[j + 1]:
                array[j], array[j + 1] = array[j + 1], array[j]
                draw_array(array, j, j + 1)
        self.status_label.config(text=f"Completed pass {i + 1} of {n - 1}")

def visualize_selection_sort(self):
    self.status_label.config(text="Running Selection Sort Visualization...")
    array = [random.randint(10, 100) for _ in range(20)]


    self.run_sort_algorithm(array, self.selection_sort)


def selection_sort(self, array, draw_array):
    for i in range(len(array)):
        min_idx = i
        for j in range(i + 1, len(array)):
            if array[j] < array[min_idx]:
                min_idx = j
        array[i], array[min_idx] = array[min_idx], array[i]
        draw_array(array, i, min_idx)

def visualize_quick_sort(self):
    self.status_label.config(text="Running Quick Sort Visualization...")
    array = [random.randint(10, 100) for _ in range(20)]
    self.run_sort_algorithm(array, self.quick_sort, 0, len(array) - 1)

def quick_sort(self, array, draw_array, low, high):
    if low < high:
        pi = self.partition(array, draw_array, low, high)
        self.quick_sort(array, draw_array, low, pi - 1)
        self.quick_sort(array, draw_array, pi + 1, high)

def partition(self, array, draw_array, low, high):
    pivot = array[high]
    i = low - 1
    for j in range(low, high):
        if array[j] <= pivot:
            i += 1
```

```
                                array[i], array[j] = array[j],
array[i]
                draw_array(array, i, j)
            array[i + 1], array[high] = array[high], array[i + 1]
            draw_array(array, i + 1, high)
            return i + 1

        def run_sort_algorithm(self, array, sort_function):
            self.canvas.delete("all")



            def draw_array(array, highlighted_index1=None,
highlighted_index2=None):
                self.canvas.delete("all")


                bar_width = 20
                max_height = max(array) if array else 1
                for i, value in enumerate(array):
                    x0 = i * bar_width
                    y0 = 350 - (value / max_height) * 300
                    x1 = x0 + bar_width
                    y1 = 350
                    color = (
                        "red"
                        if i == highlighted_index1 or i == highlighted_index2
                        else "blue"
                    )
                    self.canvas.create_rectangle(x0, y0, x1, y1, fill=color)
                    self.canvas.create_text(
                        (x0 + x1) / 2, y0 - 10, text=str(value), font=("Arial", 10)
                    )
                self.root.update_idletasks()
                time.sleep(self.speed_scale.get())

            sort_function(array, draw_array)
            self.status_label.config(text="Sorting Completed!")
```

```
def visualize_bfs(self):
    self.status_label.config(text="Running BFS Visualization...")
    self.canvas.delete("all")
    grid_size = 5
    cell_size = 70
    grid = [[0] * grid_size for _ in range(grid_size)]
    start = (0, 0)
    goal = (grid_size - 1, grid_size - 1)
    grid[goal[0]][goal[1]] = 2

    def bfs(start):


        queue = [start]
        visited = set()
        while queue:
            x, y = queue.pop(0)
            if (x, y) == goal:


                self.status_label.config(text="Goal reached!")
                return
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nx, ny = x + dx, y + dy
                if (
                    0 <= nx < grid_size
                    and 0 <= ny < grid_size
                    and (nx, ny) not in visited
                ):
                    visited.add((nx, ny))
                    queue.append((nx, ny))
                    grid[nx][ny] = 1
                    draw_grid(visited, queue)

    def draw_grid(visited, queue):
        self.canvas.delete("all")
        for i in range(grid_size):
            for j in range(grid_size):
                x0, y0 = j * cell_size, i * cell_size
```

```
x1, y1 = x0 + cell_size, y0 + cell_size
                color = "white"
                if (i, j) in visited:
                    color = "lightblue"
                elif (i, j) in queue:
                    color = "orange"
                if (i, j) == goal:
                    color = "green"
                self.canvas.create_rectangle(
                    x0, y0, x1, y1, fill=color, outline="black"
                )
            self.root.update_idletasks()


            time.sleep(self.speed_scale.get())

        bfs(start)
        self.status_label.config(text="BFS Completed")

    def visualize_dijkstra(self):
        self.status_label.config(text="Running Dijkstra's Algorithm...")


        self.canvas.delete("all")
        grid_size = 10
        cell_size = 35
        grid = [
            [random.randint(1, 10) for _ in range(grid_size)] for _ in
range(grid_size)
        ]
        start = (0, 0)
        goal = (grid_size - 1, grid_size - 1)

        def dijkstra(start, goal):
            distances = {
                node: float("infinity")
                for row in range(grid_size)
                for node in [(row, col) for col in range(grid_size)]
```

```
                }
        distances[start] = 0
    visited = set()
    pq = [(0, start)]
    parents = {start: None}
    while pq:
        current_distance, current_node = heapq.heappop(pq)
        if current_node in visited:
            continue
        visited.add(current_node)
        if current_node == goal:
            reconstruct_path(parents, goal)
            self.status_label.config(text="Dijkstra's Algorithm Completed!")
            return


        x, y = current_node
        for dx, dy in [
            (-1, 0),
            (1, 0),
            (0, -1),
            (0, 1),
        ]:
            neighbor = (x + dx, y + dy)
            if 0 <= neighbor[0] < grid_size and 0 <= neighbor[1] < grid_size:


                distance = current_distance + grid[neighbor[0]][neighbor[1]]
                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    parents[neighbor] = current_node
                    heapq.heappush(pq, (distance, neighbor))
        draw_grid(visited, current_node, goal)

def draw_grid(visited, current_node, goal):
    self.canvas.delete("all")
    for i in range(grid_size):
        for j in range(grid_size):
            x0, y0 = j * cell_size, i * cell_size
            x1, y1 = x0 + cell_size, y0 + cell_size
```

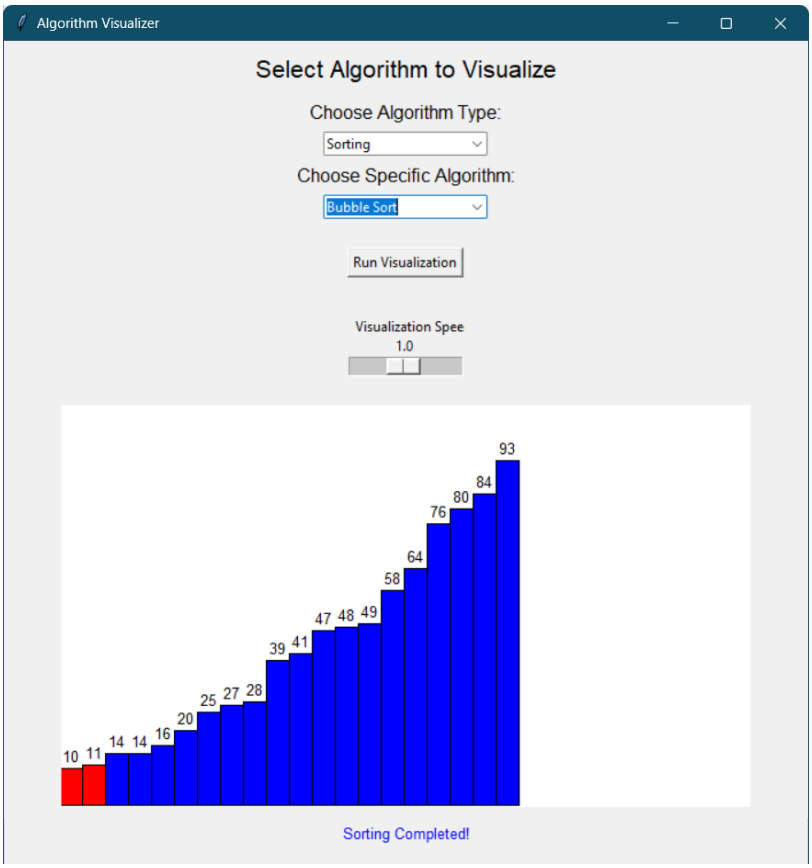
```
                color = "white"
            if (i, j) == start:
                color = "yellow"
            elif (i, j) == goal:
                color = "green"
            elif (i, j) in visited:
                color = "lightblue"
            elif (i, j) == current_node:
                color = "orange"
            self.canvas.create_rectangle(
                x0, y0, x1, y1, fill=color, outline="black"
            )
            self.canvas.create_text(
                x0 + cell_size // 2,


                y0 + cell_size // 2,
                text=str(grid[i][j]),
                font=("Arial", 8),
            )
        self.root.update_idletasks()
        time.sleep(self.speed_scale.get())

    def reconstruct_path(parents, goal):
        current = goal
        while current:
            x0, y0 = current[1] * cell_size, current[0] * cell_size


            x1, y1 = x0 + cell_size, y0 + cell_size
            self.canvas.create_rectangle(
                x0, y0, x1, y1, fill="blue", outline="black"
            )
            current = parents.get(current)
            self.root.update_idletasks()
            time.sleep(self.speed_scale.get() / 2)
    dijkstra(start, goal)

def visualize_prim(self):
```

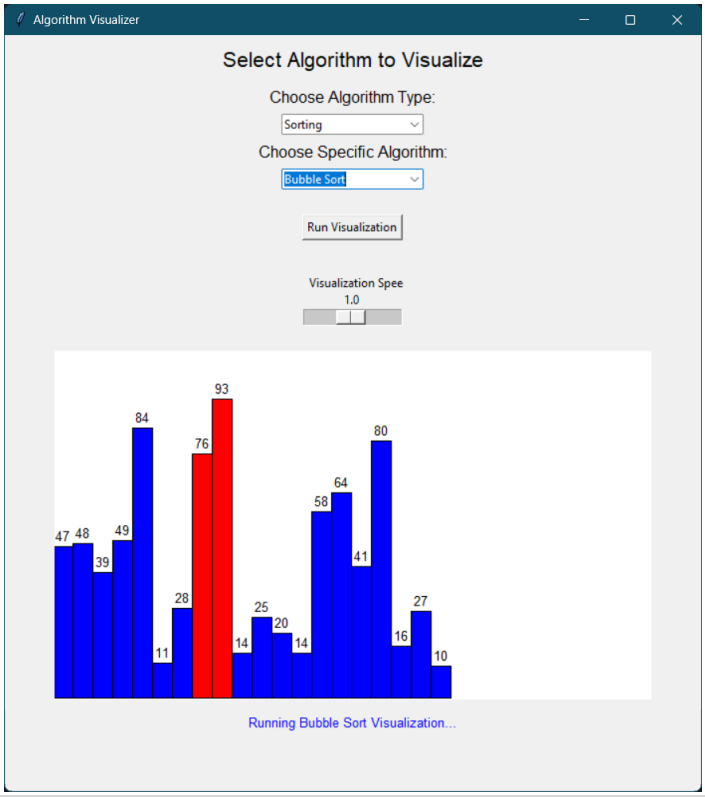
```
self.status_label.config(text="Running Prim's
Algorithm...")
        self.canvas.delete("all")
        graph = {
            0: {1: 2, 2: 3},
            1: {0: 2, 3: 1, 4: 4},
            2: {0: 3, 4: 5},
            3: {1: 1, 4: 6},
            4: {1: 4, 2: 5, 3: 6},
        }
        node_positions = {
            0: (100, 100),
            1: (200, 50),
            2: (300, 100),
            3: (200, 200),
            4: (300, 200),


        def draw_graph(mst_edges, current_edge=None, visited_nodes=set()):
            self.canvas.delete("all")
            for node, (x, y) in node_positions.items():
                color = "green" if node in visited_nodes else "white"
                self.canvas.create_oval(x - 15, y - 15, x + 15, y + 15, fill=color)
                self.canvas.create_text(x, y, text=str(node), font=("Arial", 12))
            for node, neighbors in graph.items():
                for neighbor, weight in neighbors.items():
                    x0, y0 = node_positions[node]
                    x1, y1 = node_positions[neighbor]
                    edge_color = "lightgray"


                    if (node, neighbor) in mst_edges or (neighbor, node) in mst_edges:
                        edge_color = "blue"
                    elif (node, neighbor) == current_edge or (
                        neighbor,
                        node,
                    ) == current_edge:
                        edge_color = "orange"
                    self.canvas.create_line(x0, y0, x1, y1, fill=edge_color, width=2)
```

```
                        self.canvas.create_text(
                            (x0 + x1) / 2,
                            (y0 + y1) / 2,
                            text=str(weight),
                            font=("Arial", 10),
                            fill="black",
                        )

                self.root.update_idletasks()
                time.sleep(self.speed_scale.get())

            def prim(start):
                visited_nodes = {start}
                edges = [(weight, start, to) for to, weight in graph[start].items()]
                heapq.heapify(edges)
                mst_edges = []
                while edges:


                    weight, frm, to = heapq.heappop(edges)
                    if to not in visited_nodes:
                        visited_nodes.add(to)
                        mst_edges.append((frm, to))
                        draw_graph(
                            mst_edges, current_edge=(frm, to),
visited_nodes=visited_nodes)

for next_to, next_weight in graph[to].items():
                            if next_to not in visited_nodes:
                                heapq.heappush(edges, (next_weight, to, next_to))
                self.status_label.config(text="Prim's Algorithm Completed!")
            prim(0)


if __name__ == "__main__":
    root = tk.Tk()
    app = AlgorithmVisualizerApp(root)
    root.mainloop()
```

4. Outputs

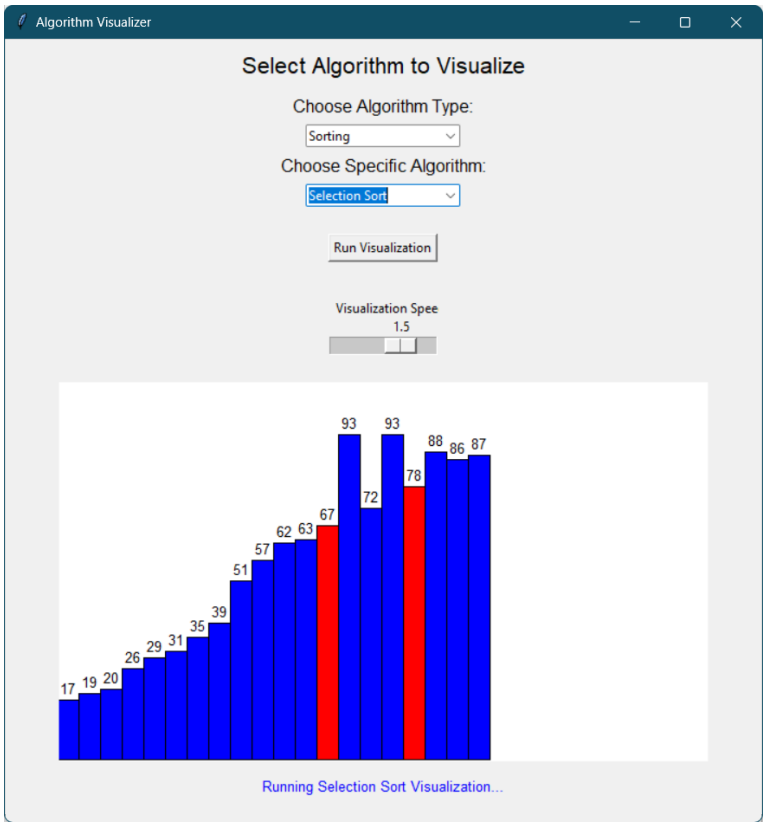
b. Completed Bubble Sorting

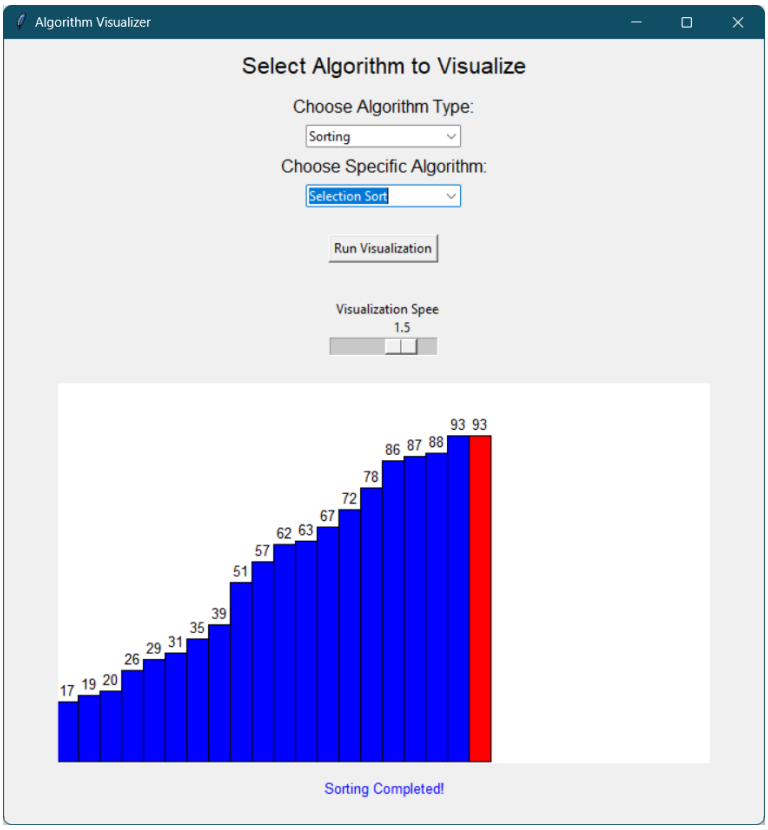




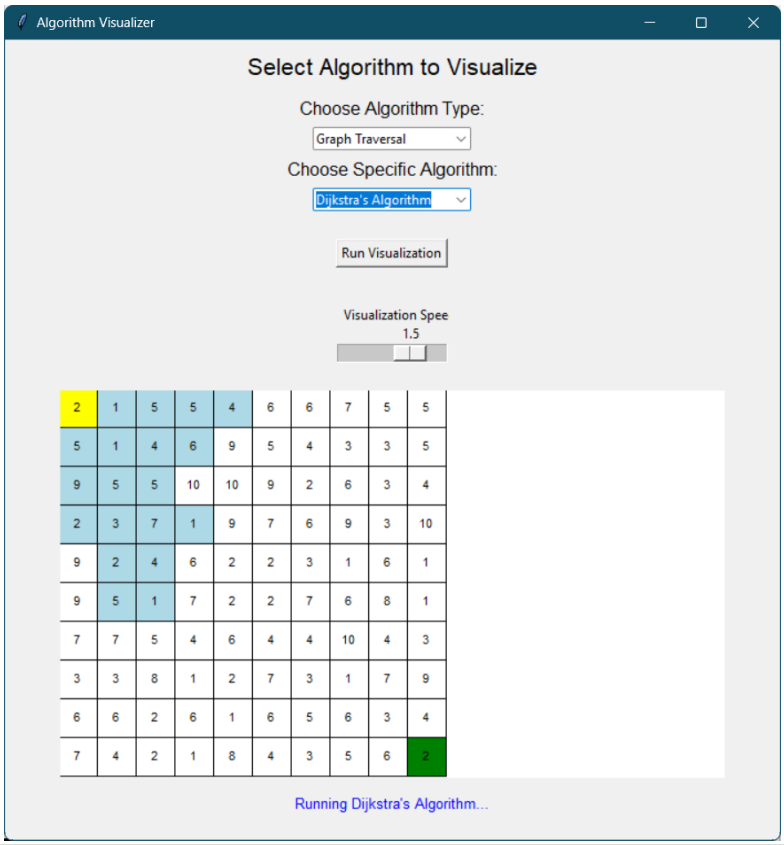
a. Performing Selection Sorting with Increased Visualization Speed

b. Performed Selection Sorting with Increased Visualization Speed



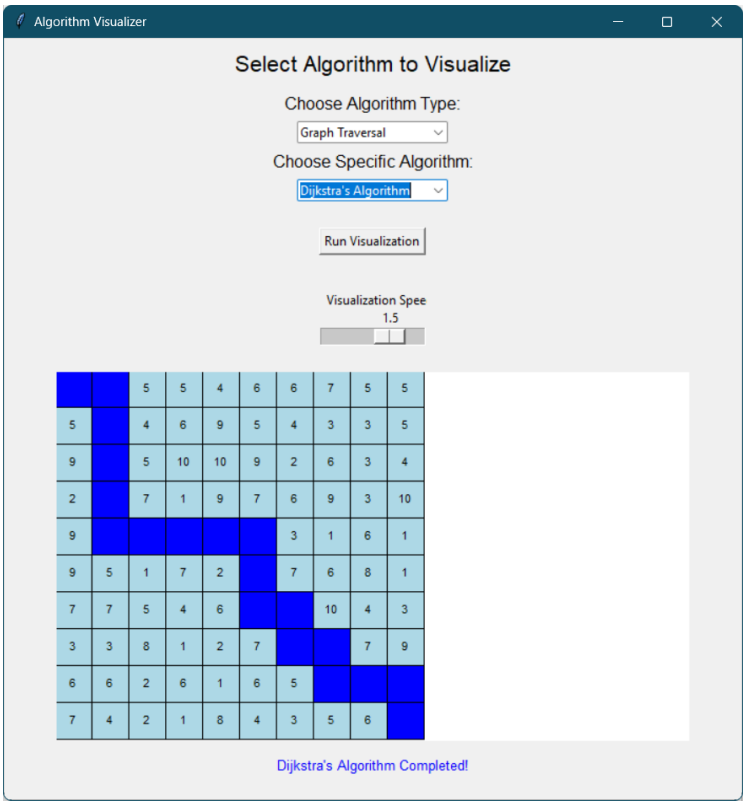


Similarly, we have some Graph Traversal Algorithms as well:

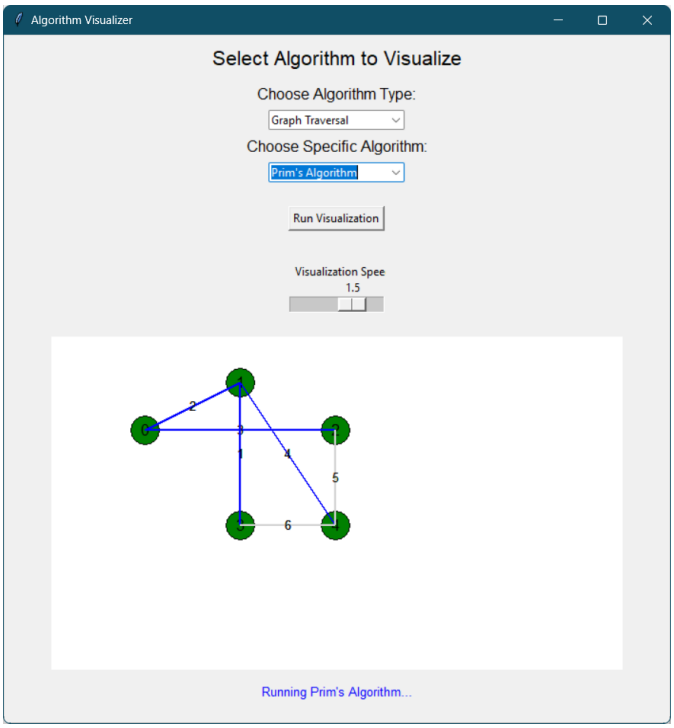


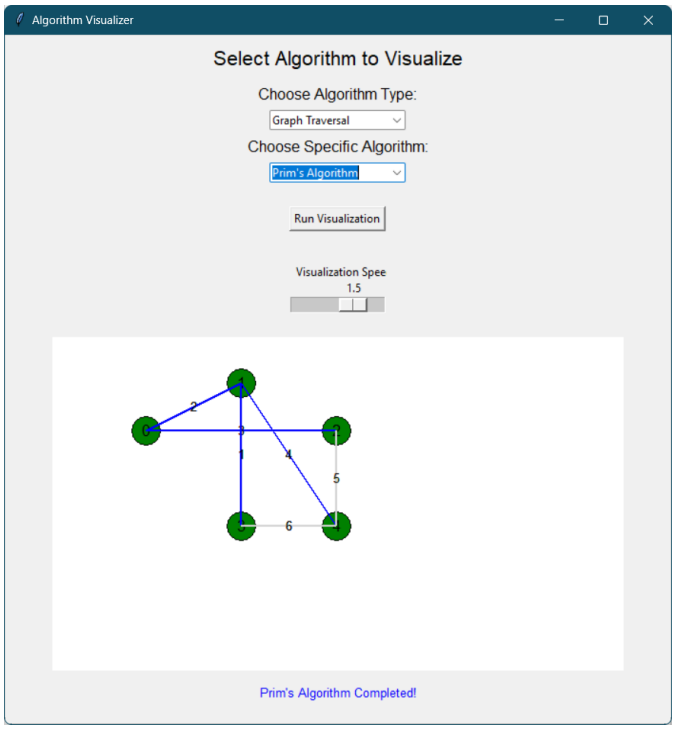
Visualizing Dijkstra's Algorithm in grid with increased speed (with increased speed)

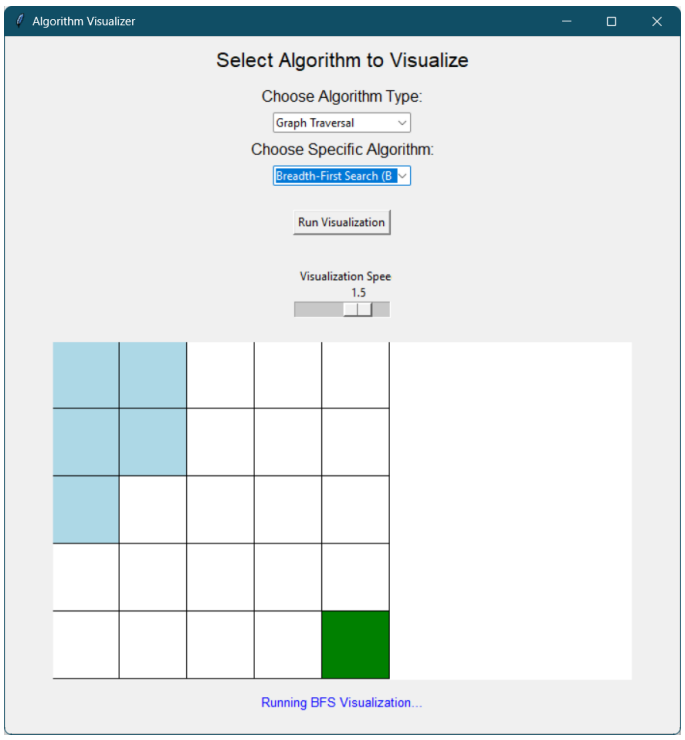
Visualized the given grid with Dijkstra's Algorithm

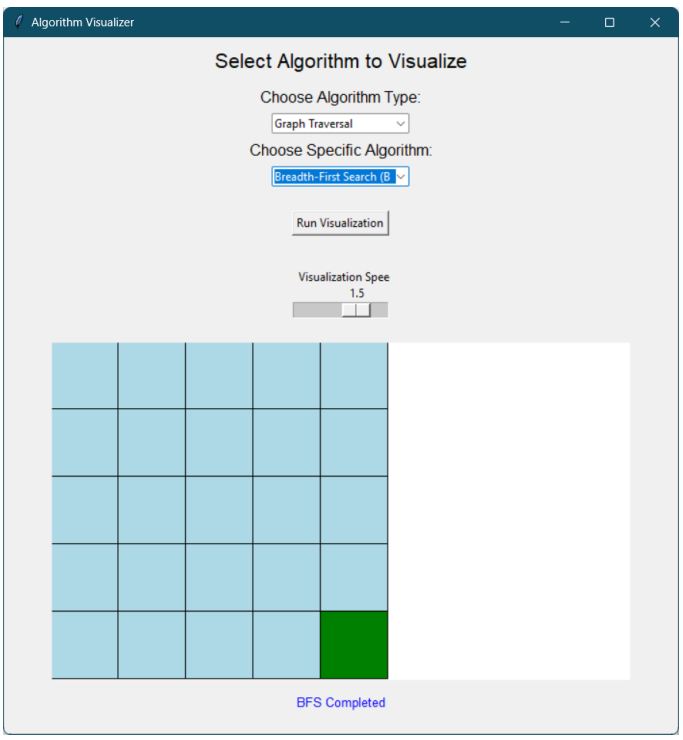


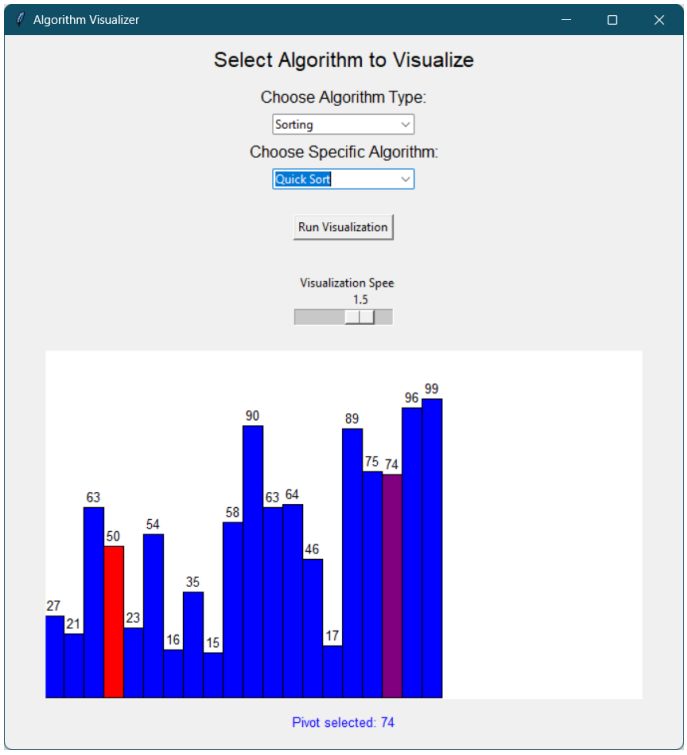
More screenshots using various Algorithms:

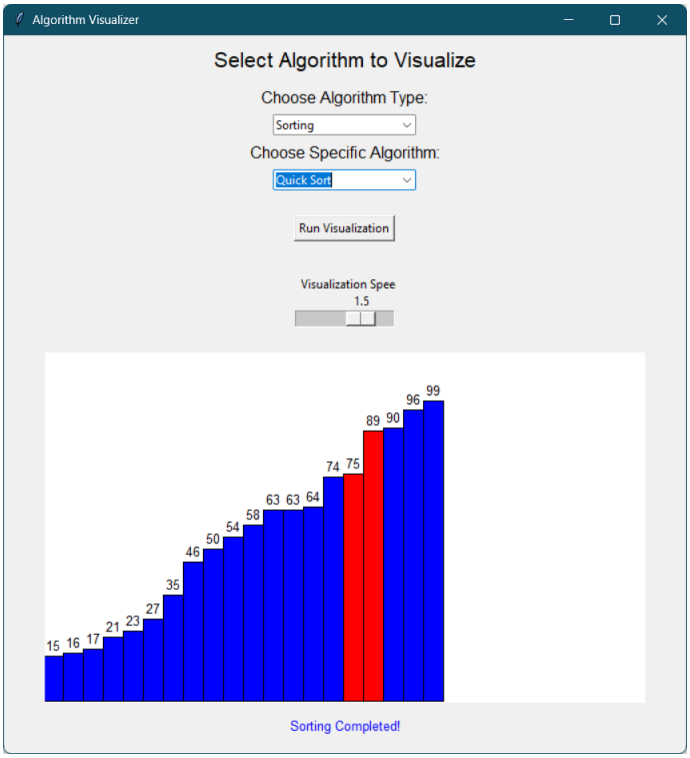





Algorithm Visualizer
Select Algorithm to Visualize
Choose Algorithm Type:
Graph Traversal
Choose Specific Algorithm:
Breadth-First Search (B
Run Visualization
Visualization Spee
1.5
Running BFS Visualization...


Algorithm Visualizer
Select Algorithm to Visualize
Choose Algorithm Type:
Graph Traversal
Choose Specific Algorithm:
Breadth-First Search (B
Run Visualization
Visualization Spee
1.5
BFS Completed


Algorithm Visualizer
Select Algorithm to Visualize
Choose Algorithm Type:
Sorting
Choose Specific Algorithm:
Quick Sort
Run Visualization
Visualization Spee
1.5
27 21 63 50 23 54 16 35 15 58 90 63 64 46 17 89 75 74 96 99
Pivot selected: 74


Algorithm Visualizer
Select Algorithm to Visualize
Choose Algorithm Type:
Sorting
Choose Specific Algorithm:
Quick Sort
Run Visualization
Visualization Spee
1.5
15 16 17 21 23 27 35 46 50 54 58 63 63 64 74 75 89 90 96 99
Sorting Completed!

5. Implementation Details

The implementation of this algorithm visualization tool utilizes Python's Tkinter library for the graphical user interface (GUI) and combines multithreading and random data generation to enhance interactivity and visualization control. Below is an outline of the core implementation components:

a. GUI Layout and User Controls

The application is structured using Tkinter to create an interactive and visually appealing interface. The main window includes a drop-down selection for algorithm type (Sorting or Graph Traversal) and specific algorithms within each type, such as Bubble Sort, Selection Sort, Quick Sort, BFS, Dijkstra's, and Prim's Algorithm. The interface also includes a speed control slider to adjust the speed of the visualization, a canvas to render animations, and a status label to display messages about the current algorithm's progress.

b. Multithreading for Responsiveness

To ensure the GUI remains responsive while algorithms are visualized, the threading library is used to run algorithms in separate threads. This avoids GUI freezes by offloading heavy computation and visualization to background threads. A decorator function, @threaded, is used to encapsulate visualization methods within separate threads, enabling the app to run the visualization smoothly alongside user interactions.

c. Sorting Algorithm Visualizations

Sorting visualizations use randomly generated arrays of integers, represented as vertical bars on the canvas. Each algorithm visualization—Bubble Sort, Selection Sort, and Quick Sort—is implemented with a corresponding visualization function. For example:

- **Bubble Sort**: Iterates through the list, swapping elements and highlighting comparisons. The draw_array function is called within each step to redraw the updated array with highlighted bars indicating the elements being compared or swapped.
- **Selection Sort**: Scans for the minimum element in each pass, swapping it with the current position. As with Bubble Sort, the draw_array function highlights changes during each iteration.
- **Quick Sort**: Partitions the array around a pivot element recursively. The partition function moves smaller elements to the left and larger ones to the right of the pivot, with draw_array illustrating the current state.

d. Graph Traversal Visualizations

Graph traversal algorithms like BFS, Dijkstra's Algorithm, and Prim's Algorithm operate on grids or node graphs. These are visually represented on the canvas with nodes and connecting edges, enabling real-time visualization of traversal progress.

- **BFS (Breadth-First Search)**: A grid-based traversal with nodes represented as cells. Each cell's state (visited or in queue) is displayed in different colors. The BFS algorithm explores neighboring nodes in layers, simulating a level-order search.
- **Dijkstra's Algorithm**: Utilizes a weighted grid where each cell has a random cost. The algorithm progressively explores the grid, using a priority queue (via heapq) to find the shortest path to the goal. Nodes on the shortest path are colored to illustrate the path reconstruction.
- **Prim's Algorithm**: A minimum spanning tree algorithm implemented with a small pre-defined graph. It selects edges with the lowest weights to form a tree, highlighting selected edges as the algorithm progresses.

e. Real-Time Visualization with draw_array and draw_grid Functions

The draw_array and draw_grid functions handle the animation of algorithm states. For sorting algorithms, draw_array redraws the bars with specific colors to indicate active comparisons. For graph traversal algorithms, draw_grid or draw_graph redraws the grid or graph, marking visited nodes, nodes in the queue, and goal nodes in distinct colors. Each frame is rendered with a brief delay using time.sleep to allow users to observe changes at each step.

f. Adjustable Visualization Speed

The speed_scale slider allows users to control the speed of animations, adjusting the sleep interval in visualization functions. This enables users to slow down or speed up visualizations based on their learning preferences.

6. System Study

The System Study of this algorithm visualization tool begins with a clear definition of the problem it aims to solve. Traditional methods for learning algorithms can feel abstract and complex, often leaving beginners without a tangible understanding of how algorithms operate. This tool provides an interactive visual representation of sorting and graph traversal algorithms, allowing users to explore these processes in real-time. By enabling step-by-step observation of algorithms like Bubble Sort, Quick Sort, BFS, and Dijkstra's Algorithm, the tool addresses the need for a more engaging, intuitive, and accessible learning method for students, educators, and self-learners alike.

The system requirements for this tool are outlined to ensure both functionality and user-friendliness. Functional requirements include options for selecting algorithm types and adjusting visualization speed, along with real-time updates on a canvas to highlight each algorithm's progression. Non-functional requirements emphasize system responsiveness and ease of use, ensuring users can interact with the tool without performance issues or complex setup. From a feasibility perspective, this tool relies on Python and Tkinter, both free and widely supported, making the system economical and operationally feasible for general users.

The interface module handles user interactions, the algorithm module defines individual algorithm functions, and the drawing module updates the canvas to reflect algorithm progress visually. Threading keeps the GUI responsive during computational tasks, ensuring a smooth user experience. By delivering a real-time, educational experience, the tool provides users with an accessible way to explore algorithms, helping to bridge the gap between theoretical learning and practical understanding. Limitations, such as scalability for larger datasets, are acknowledged but balanced by the tool's targeted focus on educational use cases.

7. Technical Feasibility

The technical feasibility of this algorithm visualization tool is well-supported by its choice of technologies, mainly Python and Tkinter, which are compatible with various platforms, including Windows, macOS, and Linux. Tkinter provides a lightweight and accessible GUI framework, making it easy to develop interactive components while ensuring cross-platform compatibility. By using Python's built-in threading capabilities, the tool maintains a responsive user interface, allowing real-time interaction and smooth visualization even during the execution of complex algorithms.

The tool relies on core Python libraries such as `time`, `random`, and `heapq`, which are efficient and add minimal system load, making the application suitable for educational use. The algorithms, including Bubble Sort, Quick Sort, BFS, and Dijkstra's Algorithm, are implemented using Python's straightforward syntax, making them easy to read and debug. Additionally, Python's popularity in educational environments means the tool is accessible to students and educators, enhancing its effectiveness as a learning resource.

Designed for moderate-sized datasets, the system is feasible for educational purposes and could be further scaled or optimized with libraries like NumPy if required for larger inputs.

8. Conclusions

In conclusion, this algorithm visualization tool successfully meets its objective of providing an interactive and intuitive platform for understanding sorting and graph traversal algorithms. Through real-time visual feedback, users can observe the inner workings of algorithms like Bubble Sort, Quick Sort, BFS, and Dijkstra's Algorithm, bridging the gap between theoretical learning and practical understanding. The choice of Python and Tkinter ensures accessibility across platforms and maintains a lightweight, user-friendly interface, suitable for students, educators, and self-learners.

The tool's modular design, leveraging threading for smooth performance, offers flexibility and responsiveness, making it well-suited for educational use. While the current version supports moderate-sized datasets and basic algorithms, the tool can be expanded with additional algorithms or optimized for larger data processing through libraries like NumPy, increasing its scope for more advanced educational settings. Overall, this visualization tool proves to be a technically feasible, accessible, and effective resource for making algorithm learning more engaging and comprehensible.

9. References

- **Tkinter Documentation. (2024).** Tkinter – *Python's GUI Toolkit.* https://docs.python.org/3/library/tkinter.html
- **Python Software Foundation. (2024).** *Python Documentation.* https://docs.python.org/3/
- **GeeksforGeeks (2024).** "Introduction to Sorting Algorithms." https://www.geeksforgeeks.org/sorting-algorithms/
- **GeeksforGeeks. (2024).** "Graph Traversal Algorithms." https://www.geeksforgeeks.org/graph-traversal-algorithms/
- **Stack Overflow. (2024).** "Python Multithreading." https://stackoverflow.com/questions/23513221/what-is-the-use-of-threading-in-python
- **Real Python. (2024).** "Creating a Simple GUI Application with Tkinter." https://realpython.com/python-gui-tkinter/