Text

Description automatically generated

**Activity based**

**Project Report on**

**Computer Network**

**Submitted to Vishwakarma University, Pune**

**Under the Initiative of**

**Contemporary Curriculum, Pedagogy, and Practice (C2P2)**

**By**

**Mudra Bhattad**

**SRN No : 202101231**

**Roll No : 23**

**Div : E**

**Third Year Engineering**

**Department of Computer Engineering**

**Faculty of Science and Technology**

**Academic Year**

**2023-2024**

**Dijkstra’s algorithm Simulation**

**Project Statement :**

Create a visualization tool for Dijkstra's algorithm to demonstrate shortest path determination in a weighted graph. It will accept multiple paths and it will highlight different paths in different colors.

**Problem Description :**

The project aims to develop a visualization tool that illustrates Dijkstra's algorithm, specifically highlighting how it computes the shortest path in a weighted graph. This interactive tool provides a user-friendly interface where users can input a graph with nodes, edges, and weights. The visualization dynamically showcases the algorithm's steps, emphasizing the shortest path determination process through graphical representation. Users can observe the algorithm's progression, node traversals, and the path discovery in real-time. This tool serves as an educational resource, aiding in the understanding and visualization of Dijkstra's algorithm and its application in finding the shortest path in a graph.

**Project Modules:**

1. Graph Input Module:

* Node Input: Allows users to input the number of nodes and their respective names.
* Edge Input: Facilitates the entry of edges along with their weights.

1. Visualization Module:

* Graph Rendering: Utilizes libraries like NetworkX and Matplotlib to render and display the graph.
* Highlight Path: Highlights the shortest path discovered by Dijkstra's algorithm.

1. Algorithm Execution:

* Dijkstra's Algorithm: Implements the logic to execute Dijkstra's algorithm on the entered graph.
* Shortest Path Determination: Identifies and highlights the shortest path found by the algorithm.

1. User Interface:

* GUI Design: Constructs the user interface to input graph details and showcase the visualization.
* Interaction Handling: Manages interactions, such as input submission and algorithm execution trigger.

1. Output Display:

* Graph Visualization: Displays the graph and its shortest path visually using appropriate libraries.
* Results Showcase: Presents the results, including the shortest path and node traversal, to the user.

**Theory:**

**Dijkstra's Algorithm:**

**Objective:** Dijkstra's algorithm finds the shortest path between nodes in a graph with non-negative edge weights.

**Approach:** It explores the graph from a source node outward, maintaining the shortest known distance to each node.

**Main Concept:** It employs a greedy strategy, iteratively selecting the node with the smallest distance from the source and updating its neighbours' distances.

**Weighted Graphs:**

**Representation:** A weighted graph consists of nodes (vertices) connected by edges (links) with associated weights or costs.

**Edge Weights:** These represent the distance, time, or any quantitative measure between nodes.

**Directed or Undirected:** Edges can be directional (directed) or bidirectional (undirected) with weights assigned accordingly.

**Visualization:**

**NetworkX and Matplotlib:** These libraries are commonly used in Python for graph visualization.

**Node-Edge Representation:** Nodes are typically represented as circles, and edges as lines connecting nodes.

**Visualization Aids:** Color-coding nodes, edges, or paths assists in distinguishing elements in the graph.

**Shortest Path:**

**Definition:** The shortest path between two nodes in a graph is the path with the minimum sum of edge weights.

**Dijkstra's Contribution:** It efficiently finds the shortest paths from one node (the source) to all other nodes in the graph.

**User Interaction:**

**Input Graph:** Users can input the graph's structure, including nodes, edges, and weights.

**Graph Manipulation:** Users can interact with the visualized graph, observe shortest paths, and modify inputs for different scenarios.

**Educational Value:**

**Learning Aid:** Visualization tools aid in understanding algorithms by providing a visual representation of their execution steps.

**Algorithm Demonstration:** Dijkstra's algorithm visualization helps in comprehending how it traverses the graph to find the shortest path.

**Complexity:**

**Time Complexity:** Dijkstra's algorithm has a time complexity of O(V^2) for naive implementations but can be optimized to O(E + V log V) using data structures like priority queues.

This theoretical knowledge forms the basis for implementing the visualization tool, guiding the understanding of Dijkstra's algorithm's principles and graph visualization techniques.

**Implementation :**

**1. User Interface Setup:**

* Design GUI: Create a user-friendly interface using a toolkit like PyQt or Tkinter.
* Input Fields: Allow users to input the number of nodes, node names, edges, and their weights.
* Graph Visualization: Display the graph and paths using NetworkX and Matplotlib.

**2. Graph Representation:**

* Graph Creation: Generate the graph structure based on user inputs (nodes, edges, weights).
* NetworkX Integration: Use NetworkX to create and manipulate the graph in Python.
* Visualization Settings: Set up the initial visualization settings (node colors, edge styles, etc.).

**3. Dijkstra's Algorithm:**

* Implementation: Code Dijkstra's algorithm to find the shortest path.
* Apply Algorithm: Use the algorithm to determine the shortest path between specified nodes.
* Update Visualization: Highlight the shortest path on the graph visualization.

**4. User Interaction and Output:**

* Input Processing: Capture user inputs for nodes, edges, and weights.
* Update Graph: After user input, update the graph display to reflect changes.
* Visualization Updates: Upon each input or algorithm execution, update the displayed graph and paths.

**5. Testing and Validation:**

* Unit Testing: Test each module or function to verify its correctness.
* Scenario Testing: Validate the tool with different graph configurations and edge cases.
* User Feedback: Gather feedback for improving user experience and functionality.

**Code :**

import sys

import networkx as nx

import matplotlib.pyplot as plt

def dijkstra(graph, source):

unvisited = set(graph.keys())

distance = {node: sys.maxsize for node in unvisited}

distance[source] = 0

parent = {}

while unvisited:

current\_node = min(unvisited, key=lambda node: distance[node])

unvisited.remove(current\_node)

for neighbor, weight in graph[current\_node].items():

if distance[current\_node] + weight < distance[neighbor]:

distance[neighbor] = distance[current\_node] + weight

parent[neighbor] = current\_node

return distance, parent

def get\_custom\_graph():

graph = {}

num\_nodes = int(input("Enter the number of nodes (up to 15): "))

if num\_nodes > 15:

print("Number of nodes exceeds the maximum limit (15).")

return None

for i in range(num\_nodes):

node\_name = input(f"Enter the name of node {i+1}: ")

graph[node\_name] = {}

while True:

connection = input("Enter a connection (e.g., 'A B 5' for a connection between A and B with weight 5) or 'done' to finish: ")

if connection == 'done':

break

parts = connection.split()

if len(parts) == 3:

node\_a, node\_b, weight = parts

if node\_a in graph and node\_b in graph:

graph[node\_a][node\_b] = int(weight)

graph[node\_b][node\_a] = int(weight)

else:

print("Invalid nodes. Please check the node names and try again.")

else:

print("Invalid input format. Please use 'A B 5' format for connections.")

return graph

custom\_graph = get\_custom\_graph()

if custom\_graph is None:

sys.exit()

def get\_valid\_node\_input(graph, prompt):

while True:

node = input(prompt)

if node in graph:

return node

else:

print("Node not found in the graph. Please check the node name and try again.")

source\_node = get\_valid\_node\_input(custom\_graph, "Enter the source node: ")

destination\_node = get\_valid\_node\_input(custom\_graph, "Enter the destination node: ")

shortest\_distances, parents = dijkstra(custom\_graph, source\_node)

G = nx.DiGraph()

for node, neighbors in custom\_graph.items():

G.add\_node(node)

for neighbor, weight in neighbors.items():

G.add\_edge(node, neighbor, weight=weight)

edge\_labels = {(node, neighbor): weight for node, neighbors in custom\_graph.items() for neighbor, weight in neighbors.items()}

shortest\_path = [destination\_node]

current\_node = destination\_node

while current\_node != source\_node:

current\_node = parents[current\_node]

shortest\_path.insert(0, current\_node)

shortest\_path\_edges = [(shortest\_path[i], shortest\_path[i + 1]) for i in range(len(shortest\_path) - 1)]

pos = nx.spring\_layout(G)

nx.draw(G, pos, with\_labels=True, node\_size=500, node\_color='lightblue', font\_size=10)

nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=edge\_labels)

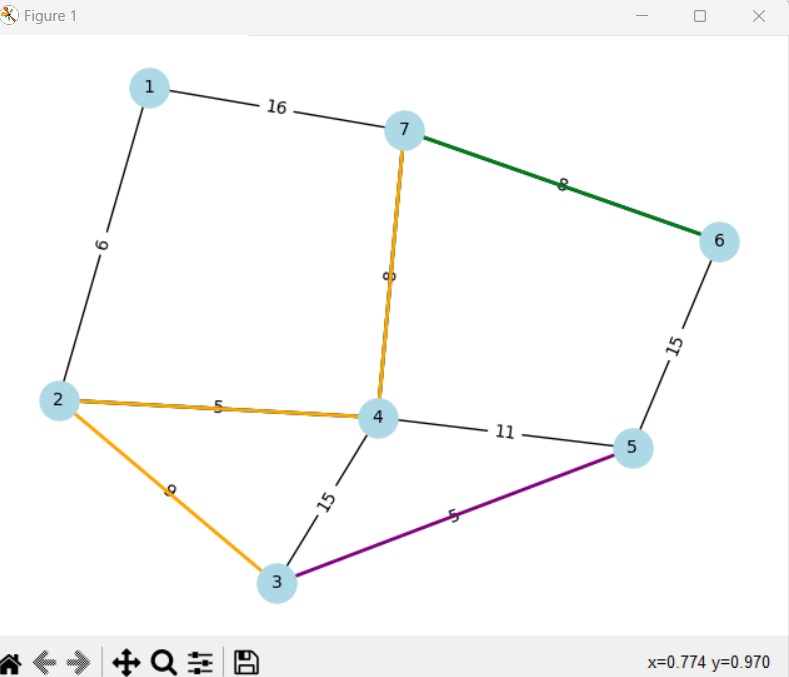
nx.draw\_networkx\_labels(G, pos, labels={source\_node: source\_node, destination\_node: destination\_node}, font\_size=10, font\_color='red')

nx.draw\_networkx\_edges(G, pos, edgelist=shortest\_path\_edges, edge\_color='red', width=2)

plt.title(f'Shortest Path from {source\_node} to {destination\_node}')

plt.show()

**Output:**



**Conclusion :**

Creating a visualization tool for Dijkstra's algorithm to demonstrate shortest path determination in weighted graphs is a valuable project. This tool allows users to interactively explore and understand how Dijkstra's algorithm finds the shortest path between nodes in a graph.

The implementation of this tool not only facilitates the visualization of graph structures but also aids in comprehending the functionality of the algorithm. Users can input custom graphs, observe how the algorithm navigates through the nodes and edges, and visualize the shortest path determined by Dijkstra's algorithm.

In conclusion, this project bridges theoretical knowledge with practical application. It provides a hands-on experience for users to understand graph traversal algorithms, offering insights into the process of finding the shortest path in graphs. Moreover, the visualization aspect enhances the educational value, making complex concepts more accessible and engaging.

\*Important Instruction

Word document Formatting:

Font: Times New Roman

Font size: for text 12 and for title : 14

Line spacing : 1.5

Give figure number with name

Give table no with name