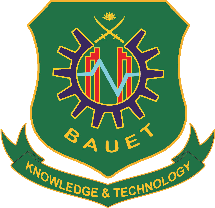
***Knowledge and Technology***

BANGLADESH ARMY UNIVERSITY OF ENGINEERING & TECHNOLOGY

**QADIRABAD, NATORE**



**DEPARTMENT OF INFORMATION AND COMMUNICATION**

**ENGINEERING**

**A project report on**

Traffic Routing in a City: Graphs, Dijkstra, BFS

Sheikh noor abdullah

&

sandipta debnath

**FALL-2024**

**May, 2025**

**Title of the Project:** Traffic Routing in a City: Graphs, Dijkstra, BFS

**Author**

Shape

**Sheikh Noor Abdullah**

**ID: 0812310205171010**

**Dept. of Information and Communication Engineering**

**Bangladesh Army University of Engineering & Technology**

**Author**

Shape

**Sandipta Debnath**

**ID: 0812310105171020**

**Dept. of Information and Communication Engineering**

**Bangladesh Army University of Engineering & Technology**

**Supervisor**

Shape

**Md. Appel Mahmud Pranto**

**Lecturer**

**Dept. of Information and Communication Engineering**

**Bangladesh Army University of Engineering & Technology**

**1. Project Overview:** Roads play Major role to the people live in various states, cities, town and villages, from day to day they travel to work, to schools, to business meetings, and to transport their goods. Even in this modern era entire world used roads, still one of the most effective mediums used most frequently for transportation and traveling. The shortest path manipulation among various points appears to be a main concern in the road networks. This project modeled a city road network with traffic flow control. This project facilitates modeling of a traffic system, adding roads, deletes roads, modifying traffic flow using code etc. In this Project I execute two Algorithms: Breadth-First Search (BFS) & Dijkstra's Algorithm. And also apply the Graph theory.

**Key Features:**

* **City Class:**
* **Graph Representation:** The City Class is represented as a graph where intersections [nodes] connected by roads [edges] with traffic weight.
* **Road Network Create:** In this Project the roads networks are automatically generated by Python Library Functions which are random.random() which is generates random float numbers between 0 & 1. Another is random.randint(a,b), this function is generate random integer values between a & b and the last function is random.randint() function. Which means the lowest and higher weight in the road, In this Project Weight means the traffic condition of a road.
* **Routing Algorithm:**
* Breadth-First Search (BFS): Find the shortest path in terms of number of intersection traversed.
* Dijkstra’s Algorithm: Find the shortest path with at least low traffic weight, offering a better and optimized route in terms of traffic routing system.
* **Visualization:**
* This project uses NetworkX & Matplotlib for visualizing the traffic roads network and the traffic design.
* This library’s created the graph ,roads and the traffic weight use the user’s input value and the code implementation.

**2.** **Language Name**: In this project I use python Programming Language.

3. **Objectives:**

* 1. To Represent a City Road Network in Graph Structure.
  2. For addition and deletion of traffic incident affecting in traffic.
  3. To Visualize the traffic road network and traffic flow using graph and color coding.
  4. To contrast Dijkstra's and BFS routes on the basis of distance and traffic.

**4. Code:**

**Traffic Routing System Code:**

**import heapq**

**import random**

**import matplotlib.pyplot as plt**

**import networkx as nx**

**from collections import deque**

**import time**

**import numpy as np**

**class City:**

**def \_\_init\_\_(self, size=10):**

**self.size = size**

**self.graph = {}**

**self.traffic\_incidents = {}**

**self.intersection\_types = {}**

**self.generate\_road\_network()**

**def generate\_road\_network(self):**

**for i in range(self.size):**

**self.graph[i] = {}**

**self.intersection\_types[i] = "major" if random.random() > 0.7 else "minor"**

**for i in range(self.size):**

**num\_connections = random.randint(2, min(4, self.size - 1))**

**possible\_connections = list(range(self.size))**

**possible\_connections.remove(i)**

**connections = random.sample(possible\_connections, num\_connections)**

**for j in connections:**

**traffic\_weight = random.randint(1, 10)**

**self.graph[i][j] = traffic\_weight**

**self.graph[j][i] = traffic\_weight**

**def add\_traffic\_incident(self, road\_start, road\_end, severity=5):**

**if road\_end in self.graph[road\_start]:**

**original\_weight = self.graph[road\_start][road\_end]**

**self.graph[road\_start][road\_end] += severity**

**self.graph[road\_end][road\_start] += severity**

**incident\_key = tuple(sorted([road\_start, road\_end]))**

**self.traffic\_incidents[incident\_key] = {**

**'original\_weight': original\_weight,**

**'severity': severity**

**}**

**print(f"Traffic incident added between intersections {road\_start} and {road\_end}")**

**return True**

**else:**

**print(f"ERROR: No road exists between intersections {road\_start} and {road\_end}")**

**return False**

**def reset\_traffic\_incident(self, road\_start, road\_end):**

**incident\_key = tuple(sorted([road\_start, road\_end]))**

**if incident\_key in self.traffic\_incidents:**

**original\_weight = self.traffic\_incidents[incident\_key]['original\_weight']**

**self.graph[road\_start][road\_end] = original\_weight**

**self.graph[road\_end][road\_start] = original\_weight**

**del self.traffic\_incidents[incident\_key]**

**print(f"Traffic incident between intersections {road\_start} and {road\_end} has been cleared")**

**return True**

**else:**

**print(f"No active traffic incident between intersections {road\_start} and {road\_end}")**

**return False**

**def reset\_all\_traffic\_incidents(self):**

**if not self.traffic\_incidents:**

**print("No active traffic incidents to clear")**

**return False**

**for (node1, node2), incident\_data in self.traffic\_incidents.items():**

**original\_weight = incident\_data['original\_weight']**

**self.graph[node1][node2] = original\_weight**

**self.graph[node2][node1] = original\_weight**

**incident\_count = len(self.traffic\_incidents)**

**self.traffic\_incidents = {}**

**print(f"All {incident\_count} traffic incidents have been cleared")**

**return True**

**def remove\_road(self, road\_start, road\_end):**

**if road\_end in self.graph[road\_start]:**

**del self.graph[road\_start][road\_end]**

**del self.graph[road\_end][road\_start]**

**incident\_key = tuple(sorted([road\_start, road\_end]))**

**if incident\_key in self.traffic\_incidents:**

**del self.traffic\_incidents[incident\_key]**

**print(f"Road between intersections {road\_start} and {road\_end} has been closed")**

**return True**

**else:**

**print(f"ERROR: No road exists between intersections {road\_start} and {road\_end}")**

**return False**

**def add\_road(self, road\_start, road\_end, traffic\_weight=5):**

**if road\_start not in self.graph or road\_end not in self.graph:**

**print(f"ERROR: One or both intersections don't exist in the city")**

**return False**

**if road\_end in self.graph[road\_start]:**

**print(f"ERROR: Road already exists between intersections {road\_start} and {road\_end}")**

**return False**

**self.graph[road\_start][road\_end] = traffic\_weight**

**self.graph[road\_end][road\_start] = traffic\_weight**

**print(f"New road added between intersections {road\_start} and {road\_end}")**

**return True**

**def adjust\_traffic\_weight(self, road\_start, road\_end, new\_weight):**

**if road\_end in self.graph[road\_start]:**

**old\_weight = self.graph[road\_start][road\_end]**

**self.graph[road\_start][road\_end] = new\_weight**

**self.graph[road\_end][road\_start] = new\_weight**

**incident\_key = tuple(sorted([road\_start, road\_end]))**

**if incident\_key in self.traffic\_incidents:**

**self.traffic\_incidents[incident\_key]['original\_weight'] = new\_weight**

**print(f"Traffic weight between intersections {road\_start} and {road\_end} updated from {old\_weight} to {new\_weight}")**

**return True**

**else:**

**print(f"ERROR: No road exists between intersections {road\_start} and {road\_end}")**

**return False**

**def bfs\_shortest\_path(self, start, end):**

**if start == end:**

**return [start]**

**visited = set()**

**queue = deque([(start, [start])])**

**while queue:**

**current, path = queue.popleft()**

**if current == end:**

**return path**

**if current not in visited:**

**visited.add(current)**

**for neighbor in self.graph[current]:**

**if neighbor not in visited:**

**queue.append((neighbor, path + [neighbor]))**

**return None # No path found**

**def dijkstra\_shortest\_path(self, start, end):**

**if start == end:**

**return [start], 0**

**pq = [(0, start, [start])]**

**visited = set()**

**while pq:**

**weight, current, path = heapq.heappop(pq)**

**if current == end:**

**return path, weight**

**if current not in visited:**

**visited.add(current)**

**for neighbor, traffic in self.graph[current].items():**

**if neighbor not in visited:**

**heapq.heappush(pq, (weight + traffic, neighbor, path + [neighbor]))**

**return None, float('inf') # No path found**

**def calculate\_path\_weight(self, path):**

**if not path or len(path) < 2:**

**return 0**

**total\_weight = 0**

**for i in range(len(path) - 1):**

**if path[i + 1] in self.graph[path[i]]:**

**total\_weight += self.graph[path[i]][path[i + 1]]**

**else:**

**return float('inf') # Path not valid**

**return total\_weight**

**def visualize(self, path1=None, path2=None, start=None, end=None, path1\_color='red', path2\_color='blue',**

**path1\_label=None, path2\_label=None, title="City Road Network with Traffic Weights"):**

**G = nx.Graph()**

**for node, neighbors in self.graph.items():**

**for neighbor, weight in neighbors.items():**

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

**pos = nx.spring\_layout(G, seed=42) # Fixed seed for consistent layout**

**plt.figure(figsize=(12, 8))**

**nx.draw\_networkx\_edges(G, pos, width=1.0, alpha=0.5)**

**path\_edges = []**

**if path1 and len(path1) > 1:**

**path1\_edges = [(path1[i], path1[i + 1]) for i in range(len(path1) - 1)]**

**nx.draw\_networkx\_edges(G, pos, edgelist=path1\_edges, width=4, edge\_color=path1\_color, label=path1\_label)**

**path\_edges.extend(path1\_edges)**

**if path2 and len(path2) > 1:**

**path2\_edges = [(path2[i], path2[i + 1]) for i in range(len(path2) - 1)]**

**nx.draw\_networkx\_edges(G, pos, edgelist=path2\_edges, width=4, edge\_color=path2\_color, label=path2\_label)**

**path\_edges.extend(path2\_edges)**

**node\_colors = []**

**node\_sizes = []**

**for node in G.nodes():**

**if node == start:**

**node\_colors.append('green')**

**node\_sizes.append(900) # Larger size for start**

**elif node == end:**

**node\_colors.append('purple')**

**node\_sizes.append(900) # Larger size for end**

**elif self.intersection\_types[node] == "major":**

**node\_colors.append('orange')**

**node\_sizes.append(700)**

**else:**

**node\_colors.append('skyblue')**

**node\_sizes.append(500)**

**nx.draw\_networkx\_nodes(G, pos, node\_size=node\_sizes, node\_color=node\_colors)**

**nx.draw\_networkx\_labels(G, pos, font\_size=12, font\_weight='bold')**

**edge\_labels = {(node, neighbor): weight for node, neighbors in self.graph.items()**

**for neighbor, weight in neighbors.items()}**

**nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=edge\_labels, font\_size=10)**

**legend\_items = []**

**if path1\_label and path1:**

**legend\_items.append(plt.Line2D([0], [0], color=path1\_color, lw=4, label=path1\_label))**

**if path2\_label and path2:**

**legend\_items.append(plt.Line2D([0], [0], color=path2\_color, lw=4, label=path2\_label))**

**legend\_items.extend([**

**plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='green', markersize=15, label='Start'),**

**plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='purple', markersize=15, label='End'),**

**plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='orange', markersize=15,label='Major Intersection'),**

**plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='skyblue', markersize=15,label='Minor Intersection')])**

**if legend\_items:**

**plt.legend(handles=legend\_items, loc='best')**

**plt.title(title)**

**plt.axis('off')**

**plt.tight\_layout()**

**plt.show()**

**def visualize\_traffic\_flow(self, paths=None, flow\_values=None):**

**G = nx.Graph()**

**for node, neighbors in self.graph.items():**

**for neighbor, weight in neighbors.items():**

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

**pos = nx.spring\_layout(G, seed=42)**

**if flow\_values is None:**

**flow\_values = {}**

**for node, neighbors in self.graph.items():**

**for neighbor, weight in neighbors.items():**

**edge = tuple(sorted([node, neighbor]))**

**if edge not in flow\_values:**

**flow\_values[edge] = max(1, 11 - weight)**

**if paths:**

**for path in paths:**

**for i in range(len(path) - 1):**

**edge = tuple(sorted([path[i], path[i + 1]]))**

**if edge in flow\_values:**

**flow\_values[edge] += 2**

**else:**

**flow\_values[edge] = 2**

**plt.figure(figsize=(12, 8))**

**node\_colors = []**

**node\_sizes = []**

**for node in G.nodes():**

**if self.intersection\_types[node] == "major":**

**node\_colors.append('orange')**

**node\_sizes.append(700)**

**else:**

**node\_colors.append('skyblue')**

**node\_sizes.append(500)**

**nx.draw\_networkx\_nodes(G, pos, node\_size=node\_sizes, node\_color=node\_colors)**

**for edge in G.edges():**

**sorted\_edge = tuple(sorted(edge))**

**flow = flow\_values.get(sorted\_edge, 1)**

**weight = self.graph[edge[0]][edge[1]]**

**if weight >= 8:**

**color = 'red'**

**elif weight >= 5:**

**color = 'orange'**

**else:**

**color = 'green'**

**nx.draw\_networkx\_edges(G, pos, edgelist=[edge], width=flow,edge\_color=color, alpha=0.7)**

**nx.draw\_networkx\_labels(G, pos, font\_size=12, font\_weight='bold')**

**edge\_labels = {(node, neighbor): weight for node, neighbors in self.graph.items()**

**for neighbor, weight in neighbors.items()}**

**nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=edge\_labels, font\_size=10)**

**legend\_items = [**

**plt.Line2D([0], [0], color='green', lw=4, label='Light Traffic (1-4)'),**

**plt.Line2D([0], [0], color='orange', lw=4, label='Medium Traffic (5-7)'),**

**plt.Line2D([0], [0], color='red', lw=4, label='Heavy Traffic (8-10)')]**

**legend\_items.extend([**

**plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='orange', markersize=15,**

**label='Major Intersection'),**

**plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='skyblue', markersize=15,**

**label='Minor Intersection')])**

**plt.legend(handles=legend\_items, loc='best')**

**plt.title("City Traffic Flow Visualization")**

**plt.axis('off')**

**plt.tight\_layout()**

**plt.show()**

**def compare\_algorithms(city, start, end):**

**print(f"\nRouting from intersection {start} to {end}:")**

**bfs\_start\_time = time.time()**

**bfs\_path = city.bfs\_shortest\_path(start, end)**

**bfs\_time = time.time() - bfs\_start\_time**

**bfs\_weight = city.calculate\_path\_weight(bfs\_path)**

**dijkstra\_start\_time = time.time()**

**dijkstra\_path, dijkstra\_weight = city.dijkstra\_shortest\_path(start, end)**

**dijkstra\_time = time.time() - dijkstra\_start\_time**

**print("\nBFS Path (fewest intersections):")**

**print(f" Path: {bfs\_path}")**

**print(f" Number of intersections: {len(bfs\_path)}")**

**print(f" Total traffic weight: {bfs\_weight}")**

**print(f" Computation time: {bfs\_time:.6f} seconds")**

**print("\nDijkstra Path (minimum traffic):")**

**print(f" Path: {dijkstra\_path}")**

**print(f" Number of intersections: {len(dijkstra\_path)}")**

**print(f" Total traffic weight: {dijkstra\_weight}")**

**print(f" Computation time: {dijkstra\_time:.6f} seconds")**

**optimal\_path, optimal\_reason = determine\_optimal\_path(**

**bfs\_path, bfs\_weight, bfs\_time,**

**dijkstra\_path, dijkstra\_weight, dijkstra\_time)**

**print(f"\nOptimal Path Recommendation: {optimal\_reason}")**

**print("\nVisualizing both paths for comparison:")**

**city.visualize(**

**path1=bfs\_path,**

**path2=dijkstra\_path,**

**start=start,**

**end=end,**

**path1\_color='blue',**

**path2\_color='red',**

**path1\_label='BFS Path',**

**path2\_label='Dijkstra Path'**

**)**

**return bfs\_path, dijkstra\_path, optimal\_path**

**def determine\_optimal\_path(bfs\_path, bfs\_weight, bfs\_time,dijkstra\_path, dijkstra\_weight, dijkstra\_time):**

**path\_diff = len(dijkstra\_path) - len(bfs\_path)**

**weight\_diff = bfs\_weight - dijkstra\_weight**

**if bfs\_path == dijkstra\_path:**

**return bfs\_path, "Both algorithms found the same path, which is optimal for both distance and traffic."**

**if path\_diff <= 0:**

**return dijkstra\_path, "Dijkstra's path is optimal - it has both minimal traffic and shortest distance."**

**if weight\_diff <= 0:**

**return bfs\_path, "BFS path is optimal - it has both minimal distance and lowest traffic."**

**pct\_longer\_path = path\_diff / len(bfs\_path) \* 100**

**pct\_heavier\_traffic = weight\_diff / dijkstra\_weight \* 100**

**if pct\_longer\_path > 30 and pct\_heavier\_traffic < 20:**

**return bfs\_path, f"BFS path is optimal - {pct\_longer\_path:.1f}% shorter route with only {pct\_heavier\_traffic:.1f}% more traffic."**

**elif pct\_heavier\_traffic > 40:**

**return dijkstra\_path, f"Dijkstra's path is optimal - reduces traffic by {pct\_heavier\_traffic:.1f}% with only {pct\_longer\_path:.1f}% longer distance."**

**elif pct\_longer\_path < 15:**

**return dijkstra\_path, f"Dijkstra's path is optimal - only {pct\_longer\_path:.1f}% longer but reduces traffic by {pct\_heavier\_traffic:.1f}%."**

**else:**

**if any(edge in dijkstra\_path for edge in range(len(dijkstra\_path) - 1) if**

**dijkstra\_path[edge] in bfs\_path and dijkstra\_path[edge + 1] in bfs\_path and**

**bfs\_path.index(dijkstra\_path[edge + 1]) - bfs\_path.index(dijkstra\_path[edge]) == 1):**

**return dijkstra\_path, f"Dijkstra's path is optimal - strategically avoids heavy traffic areas with only {pct\_longer\_path:.1f}% longer route."**

**else:**

**return bfs\_path, f"BFS path is optimal - significantly shorter route ({pct\_longer\_path:.1f}% difference) and traffic difference ({pct\_heavier\_traffic:.1f}%) is acceptable."**

**def simulate\_traffic\_changes(city, start, end):**

**print("\n=== Initial Route ===")**

**initial\_path, initial\_weight = city.dijkstra\_shortest\_path(start, end)**

**print(f"Initial optimal path: {initial\_path}")**

**print(f"Initial traffic weight: {initial\_weight}")**

**city.visualize(path1=initial\_path, start=start, end=end, path1\_label="Initial Route")**

**if len(initial\_path) > 2:**

**incident\_start = initial\_path[1]**

**incident\_end = initial\_path[2]**

**print("\n=== Adding Traffic Incident ===")**

**city.add\_traffic\_incident(incident\_start, incident\_end, severity=8)**

**new\_path, new\_weight = city.dijkstra\_shortest\_path(start, end)**

**print(f"New optimal path: {new\_path}")**

**print(f"New traffic weight: {new\_weight}")**

**city.visualize(**

**path1=initial\_path,**

**path2=new\_path,**

**start=start,**

**end=end,**

**path1\_color='orange',**

**path2\_color='green',**

**path1\_label='Original Route',**

**path2\_label='Re-routed Path'**

**)**

**print("\n=== Resetting Traffic Incident ===")**

**city.reset\_traffic\_incident(incident\_start, incident\_end)**

**reset\_path, reset\_weight = city.dijkstra\_shortest\_path(start, end)**

**print(f"Route after incident cleared: {reset\_path}")**

**print(f"Traffic weight after incident cleared: {reset\_weight}")**

**city.visualize(path1=reset\_path, start=start, end=end, path1\_label="Restored Route")**

**def simulate\_complex\_incident(city):**

**print("\n=== Simulating Complex Traffic Incident Scenario ===")**

**source = random.randint(0, city.size - 1)**

**dest = random.randint(0, city.size - 1)**

**while dest == source:**

**dest = random.randint(0, city.size - 1)**

**initial\_path, initial\_weight = city.dijkstra\_shortest\_path(source, dest)**

**print(f"\nInitial optimal path from {source} to {dest}: {initial\_path}")**

**print(f"Initial traffic weight: {initial\_weight}")**

**city.visualize(**

**path1=initial\_path,**

**start=source,**

**end=dest,**

**path1\_color='blue',**

**path1\_label='Initial Path',**

**title="Initial Route Before Incidents"**

**)**

**print("\n=== Adding Multiple Traffic Incidents ===")**

**incidents = []**

**if len(initial\_path) > 2:**

**incident1\_start = initial\_path[1]**

**incident1\_end = initial\_path[2]**

**severity1 = random.randint(5, 10)**

**if city.add\_traffic\_incident(incident1\_start, incident1\_end, severity1):**

**incidents.append((incident1\_start, incident1\_end))**

**print(f"Incident 1: Major accident between intersections {incident1\_start} and {incident1\_end}")**

**available\_edges = [(u, v) for u in city.graph for v in city.graph[u] if u < v and (u, v) not in incidents]**

**if available\_edges:**

**incident2\_start, incident2\_end = random.choice(available\_edges)**

**severity2 = random.randint(3, 8)**

**if city.add\_traffic\_incident(incident2\_start, incident2\_end, severity2):**

**incidents.append((incident2\_start, incident2\_end))**

**print(f"Incident 2: Construction work between intersections {incident2\_start} and {incident2\_end}")**

**available\_edges = [(u, v) for u in city.graph for v in city.graph[u] if u < v and (u, v) not in incidents]**

**if available\_edges:**

**incident3\_start, incident3\_end = random.choice(available\_edges)**

**if city.remove\_road(incident3\_start, incident3\_end):**

**print(f"Incident 3: Road closure between intersections {incident3\_start} and {incident3\_end}")**

**new\_path, new\_weight = city.dijkstra\_shortest\_path(source, dest)**

**print(f"\nNew optimal path after incidents: {new\_path}")**

**print(f"New traffic weight: {new\_weight}")**

**weight\_difference = new\_weight - initial\_weight**

**path\_difference = len(new\_path) - len(initial\_path)**

**print(f"\nDetour analysis:")**

**print(f" Increase in traffic weight: {weight\_difference} ({(weight\_difference / initial\_weight \* 100):.1f}%)")**

**print(f" Additional intersections: {path\_difference}")**

**city.visualize(**

**path1=initial\_path,**

**path2=new\_path,**

**start=source,**

**end=dest,**

**path1\_color='gray',**

**path2\_color='red',**

**path1\_label='Original Path (Now Blocked)',**

**path2\_label='New Detour Path',**

**title="Complex Incident Scenario - Detour Path" )**

**if weight\_difference > 10 or path\_difference > 3:**

**print(**

**"\nRECOMMENDATION: Deploy traffic controllers at major intersections to manage increased flow on detour route.")**

**else:**

**print("\nRECOMMENDATION: Standard traffic management sufficient for this detour scenario.")**

**print("\n=== Clearing All Traffic Incidents ===")**

**city.reset\_all\_traffic\_incidents()**

**final\_path, final\_weight = city.dijkstra\_shortest\_path(source, dest)**

**print(f"\nPath after clearing incidents: {final\_path}")**

**print(f"Traffic weight after clearing incidents: {final\_weight}")**

**city.visualize(**

**path1=final\_path,**

**start=source,**

**end=dest,**

**path1\_color='green',**

**path1\_label='Restored Path',**

**title="Routes After Clearing Incidents" )**

**def simulate\_traffic\_flow(city):**

**print("\n=== Simulating City-wide Traffic Flow ===")**

**num\_routes = min(15, city.size \* 2)**

**routes = []**

**print(f"Simulating {num\_routes} active routes across the city...")**

**for \_ in range(num\_routes):**

**source = random.randint(0, city.size - 1)**

**dest = random.randint(0, city.size - 1)**

**if source == dest:**

**continue**

**path, \_ = city.dijkstra\_shortest\_path(source, dest)**

**if path:**

**routes.append(path)**

**flow\_values = {}**

**for node, neighbors in city.graph.items():**

**for neighbor, weight in neighbors.items():**

**edge = tuple(sorted([node, neighbor]))**

**if edge not in flow\_values:**

**flow\_values[edge] = max(1, (11 - weight) / 2)**

**for path in routes:**

**for i in range(len(path) - 1):**

**edge = tuple(sorted([path[i], path[i + 1]]))**

**if edge in flow\_values:**

**flow\_values[edge] += 1.5 # Increase flow for actual routes**

**else: flow\_values[edge] = 1.5**

**congestion\_threshold = 3.0**

**hotspots = {edge: flow for edge, flow in flow\_values.items()**

**if flow > congestion\_threshold and city.graph[edge[0]][edge[1]] > 6}**

**print("\nTraffic Flow Analysis:")**

**print(f" Total active routes: {len(routes)}")**

**print(f" Congestion hotspots detected: {len(hotspots)}")**

**if hotspots:**

**print("\nTop congestion hotspots:")**

**sorted\_hotspots = sorted(hotspots.items(), key=lambda x: x[1], reverse=True)[:3]**

**for (u, v), flow in sorted\_hotspots:**

**print(f" Intersections {u} and {v}: Flow value {flow:.1f}, Traffic weight {city.graph[u][v]}")**

**print("\nVisualizing city-wide traffic flow...")**

**city.visualize\_traffic\_flow(paths=routes, flow\_values=flow\_values)**

**print("\nTime-based Traffic Simulation:")**

**times = ["Morning Rush Hour (7-9 AM)",**

**"Midday (11 AM - 1 PM)",**

**"Evening Rush Hour (4-6 PM)"]**

**for time\_period in times:**

**print(f"\n{time\_period} Traffic Pattern:")**

**time\_flow = {}**

**for edge, flow in flow\_values.items():**

**if "Morning" in time\_period:**

**if edge[0] > edge[1]:**

**time\_flow[edge] = flow \* 1.5**

**else:**

**time\_flow[edge] = flow \* 0.8**

**elif "Evening" in time\_period:**

**if edge[0] < edge[1]:**

**time\_flow[edge] = flow \* 1.5**

**else:**

**time\_flow[edge] = flow \* 0.8**

**else:**

**time\_flow[edge] = flow \* random.uniform(0.8, 1.2)**

**time\_hotspots = {edge: flow for edge, flow in time\_flow.items()**

**if flow > congestion\_threshold and city.graph[edge[0]][edge[1]] > 6}**

**print(f" Congestion hotspots: {len(time\_hotspots)}")**

**city.visualize\_traffic\_flow(flow\_values=time\_flow)**

**def user\_input\_mode():**

**print("\n=== Traffic Routing Interactive Mode ===")**

**while True:**

**try:**

**size = int(input("\nEnter the number of intersections for the city (5-20): "))**

**if 5 <= size <= 20:**

**break**

**else:**

**print("Please enter a number between 5 and 20.")**

**except ValueError:**

**print("Please enter a valid number.")**

**city = City(size=size)**

**print(f"\nCreated city with {size} intersections.")**

**city.visualize()**

**while True:**

**print("\n=== MENU ===")**

**print("1. Find route between intersections")**

**print("2. Add traffic incident")**

**print("3. Reset specific traffic incident")**

**print("4. Reset all traffic incidents")**

**print("5. Add new road")**

**print("6. Remove road")**

**print("7. Adjust traffic weight")**

**print("8. View city map")**

**print("9. Simulate traffic flow")**

**print("10. Compare routing algorithms")**

**print("11. Simulate changing traffic conditions")**

**print("12. Modify city layout")**

**print("13. Simulate complex incident")**

**print("14. Determine optimal path")**

**print("15. Exit")**

**choice = input("\nEnter your choice (1-15): ")**

**if choice == '1':**

**try:**

**start = int(input(f"Enter starting intersection (0-{size - 1}): "))**

**end = int(input(f"Enter destination intersection (0-{size - 1}): "))**

**if 0 <= start < size and 0 <= end < size:**

**compare\_algorithms(city, start, end)**

**else:**

**print(f"Intersections must be between 0 and {size - 1}.")**

**except ValueError:**

**print("Please enter valid intersection numbers.")**

**elif choice == '2':**

**try:**

**start = int(input(f"Enter first intersection of road (0-{size - 1}): "))**

**end = int(input(f"Enter second intersection of road (0-{size - 1}): "))**

**severity = int(input("Enter incident severity (1-10): "))**

**if 0 <= start < size and 0 <= end < size and 1 <= severity <= 10:**

**city.add\_traffic\_incident(start, end, severity)**

**city.visualize(start=start, end=end)**

**else:**

**print("Invalid input values.")**

**except ValueError:**

**print("Please enter valid numbers.")**

**elif choice == '3':**

**try:**

**start = int(input(f"Enter first intersection of incident road (0-{size - 1}): "))**

**end = int(input(f"Enter second intersection of incident road (0-{size - 1}): "))**

**if 0 <= start < size and 0 <= end < size:**

**city.reset\_traffic\_incident(start, end)**

**city.visualize()**

**else:**

**print(f"Intersections must be between 0 and {size - 1}.")**

**except ValueError:**

**print("Please enter valid intersection numbers.")**

**elif choice == '4':**

**city.reset\_all\_traffic\_incidents()**

**city.visualize()**

**elif choice == '5':**

**try:**

**start = int(input(f"Enter first intersection for new road (0-{size - 1}): "))**

**end = int(input(f"Enter second intersection for new road (0-{size - 1}): "))**

**weight = int(input("Enter traffic weight for new road (1-10): "))**

**if 0 <= start < size and 0 <= end < size and 1 <= weight <= 10:**

**city.add\_road(start, end, weight)**

**city.visualize()**

**else:**

**print("Invalid input values.")**

**except ValueError:**

**print("Please enter valid numbers.")**

**elif choice == '6':**

**try:**

**start = int(input(f"Enter first intersection of road to remove (0-{size - 1}): "))**

**end = int(input(f"Enter second intersection of road to remove (0-{size - 1}): "))**

**if 0 <= start < size and 0 <= end < size:**

**city.remove\_road(start, end)**

**city.visualize()**

**else:**

**print(f"Intersections must be between 0 and {size - 1}.")**

**except ValueError:**

**print("Please enter valid intersection numbers.")**

**elif choice == '7':**

**try:**

**start = int(input(f"Enter first intersection of road (0-{size - 1}): "))**

**end = int(input(f"Enter second intersection of road (0-{size - 1}): "))**

**weight = int(input("Enter new traffic weight (1-10): "))**

**if 0 <= start < size and 0 <= end < size and 1 <= weight <= 10:**

**city.adjust\_traffic\_weight(start, end, weight)**

**city.visualize()**

**else:**

**print("Invalid input values.")**

**except ValueError:**

**print("Please enter valid numbers.")**

**elif choice == '8':**

**city.visualize()**

**elif choice == '15':**

**print("Exiting program. Goodbye!")**

**break**

**elif choice == '10':**

**try:**

**start = int(input(f"Enter starting intersection (0-{size - 1}): "))**

**end = int(input(f"Enter destination intersection (0-{size - 1}): "))**

**if 0 <= start < size and 0 <= end < size:**

**compare\_algorithms(city, start, end)**

**else:**

**print(f"Intersections must be between 0 and {size - 1}.")**

**except ValueError:**

**print("Please enter valid intersection numbers.")**

**elif choice == '11':**

**try:**

**start = int(input(f"Enter starting intersection (0-{size - 1}): "))**

**end = int(input(f"Enter destination intersection (0-{size - 1}): "))**

**if 0 <= start < size and 0 <= end < size:**

**simulate\_traffic\_changes(city, start, end)**

**else:**

**print(f"Intersections must be between 0 and {size - 1}.")**

**except ValueError:**

**print("Please enter valid intersection numbers.")**

**elif choice == '12':**

**print("\nModifying city layout...")**

**# Create a few random new roads**

**for \_ in range(2):**

**start = random.randint(0, size - 1)**

**end = random.randint(0, size - 1)**

**if start != end:**

**city.add\_road(start, end, random.randint(1, 10))**

**edges = [(u, v) for u in city.graph for v in city.graph[u] if u < v]**

**if edges:**

**edge = random.choice(edges)**

**city.remove\_road(edge[0], edge[1])**

**print("City layout modified with new and removed roads.")**

**city.visualize()**

**elif choice == '13':**

**simulate\_complex\_incident(city)**

**elif choice == '14':**

**try:**

**start = int(input(f"Enter starting intersection (0-{size - 1}): "))**

**end = int(input(f"Enter destination intersection (0-{size - 1}): "))**

**if 0 <= start < size and 0 <= end < size:**

**bfs\_path = city.bfs\_shortest\_path(start, end)**

**bfs\_weight = city.calculate\_path\_weight(bfs\_path)**

**bfs\_time = 0**

**dijkstra\_path, dijkstra\_weight = city.dijkstra\_shortest\_path(start, end)**

**dijkstra\_time = 0**

**optimal\_path, reason = determine\_optimal\_path(**

**bfs\_path, bfs\_weight, bfs\_time,**

**dijkstra\_path, dijkstra\_weight, dijkstra\_time)**

**print(f"\nOptimal route recommendation: {reason}")**

**city.visualize(**

**path1=optimal\_path,**

**start=start,**

**end=end,**

**path1\_color='green',**

**path1\_label='Optimal Path')**

**else:**

**print(f"Intersections must be between 0 and {size - 1}.")**

**except ValueError:**

**print("Please enter valid intersection numbers.")**

**elif choice == '9':**

**simulate\_traffic\_flow(city)**

**else:**

**print("Invalid choice. Please enter a number between 1 and 15.")**

**def main():**

**print("=== Traffic Routing in a City ===")**

**print("1. Run demo with preset scenarios")**

**print("2. Enter interactive mode")**

**choice = input("\nEnter your choice (1-2): ")**

**if choice == '1':**

**print("\nCreating city road network...")**

**city = City(size=10)**

**print("\nInitial city road network:")**

**city.visualize()**

**start\_intersection = 0**

**end\_intersection = 9**

**compare\_algorithms(city, start\_intersection, end\_intersection)**

**simulate\_traffic\_changes(city, start\_intersection, end\_intersection)**

**simulate\_complex\_incident(city)**

**simulate\_traffic\_flow(city)**

**print("\nDemonstrating additional features:")**

**new\_road\_start = 2**

**new\_road\_end = 8**

**city.add\_road(new\_road\_start, new\_road\_end, traffic\_weight=2)**

**road\_to\_close\_start = 4**

**road\_to\_close\_end = 5**

**city.remove\_road(road\_to\_close\_start, road\_to\_close\_end)**

**print("\nCity road network after modifications:")**

**city.visualize()**

**compare\_algorithms(city, start\_intersection, end\_intersection)**

**elif choice == '2':**

**user\_input\_mode()**

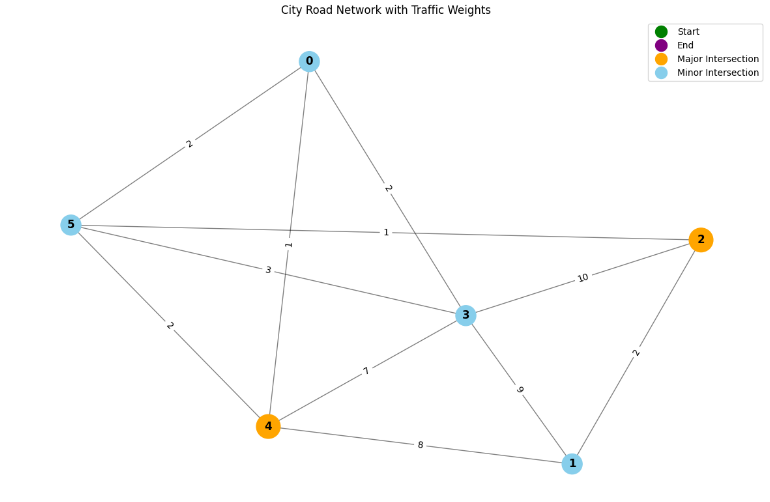
**else:**

**print("Invalid choice. Exiting program.")**

**if \_\_name\_\_ == "\_\_main\_\_":**

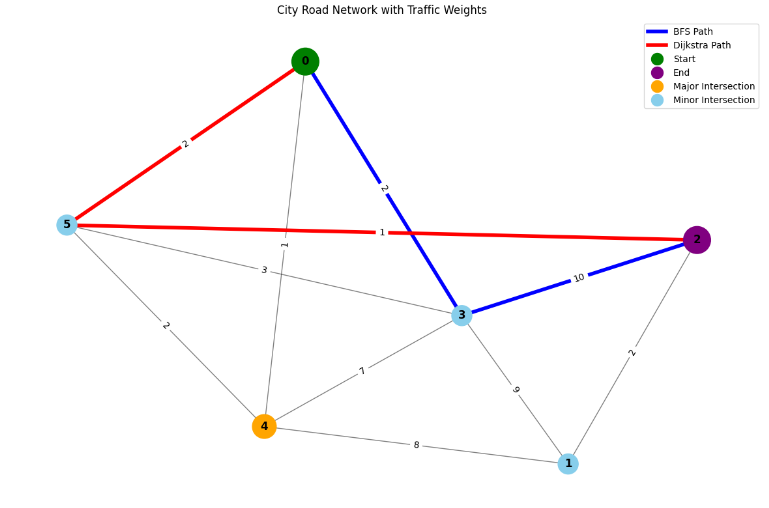
**main()**

**5. Result Analysis:**

****

**Figure 1.1: City Road Network Traffic**

**1.1 Explain:** This Graph has 6 Vertex and 11 edges. The Vertex means the interaction , the edges means how many roads in this graph and the weight means the traffic condition in this graph. If the weight is 7-10 then the traffic is very high.

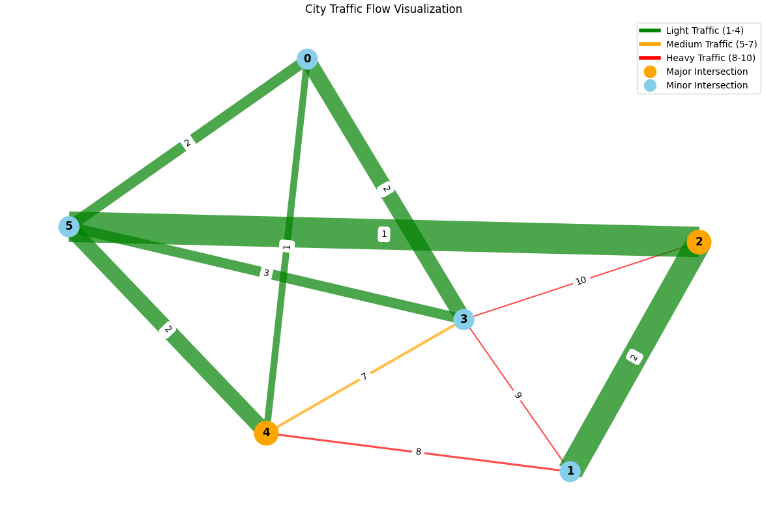
****

**Figure 1.2: City Road Network with Traffic (Route Between the intersection)**

**1.2 Explain:** The **Bule** is BFS Path and the **Red** Line is **Dijkstra** Path**.**

* **If we calculate the paths:**
* **BFS = 2 + 10 = 12**
* **Dijkstra = 2 + 1 = 3**

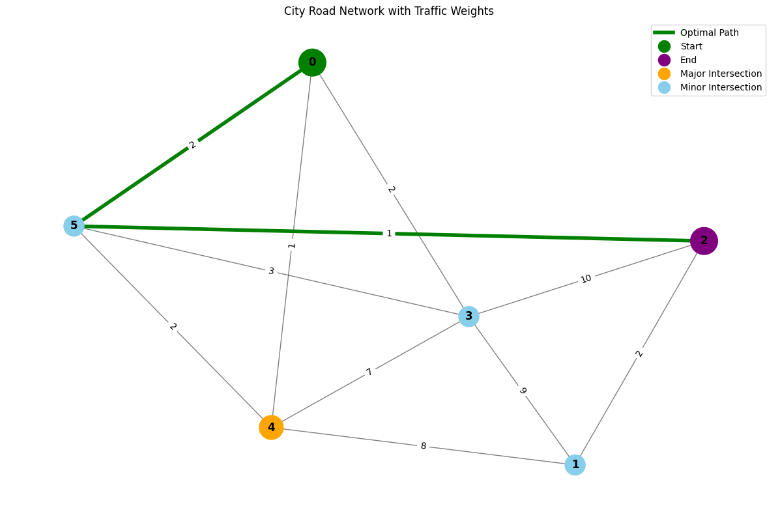
This Calculation means that the Dijkstra path is the optimal path for this graph, Which takes minimum traffic.

****

**Figure 1.3: City traffic flow Visualization**

**1.3 Explain:** In this graph the:

* [1–4] this means the Light traffic
* [5-7] this means Medium traffic
* [8-10] this means Heavy traffic.

****

**Figure 1.4: Optimal Path of City Roads network.**

**1.4 Explain:** In thisfunctionreceive multiple parameter path , weight and time of both BFS & Dijkstra Algorithm and compare those algorithm and then return the optimal path.

**6. Project Contribution:**

1. **Sheikh Noor Abdullah:**

* **Technical Development:** Was the project lead in designing a traffic routing simulation system, including:

City Road Network Generation: Implemented algorithms to simulate city road infrastructures dynamically.

* **Traffic Incident Management:** Installed systems to simulate and manage real-time traffic incidents.

Pathfinding Algorithms: Utilized BFS (Breadth-First Search) and Dijkstra's algorithms to compute optimal routes.

* **Visualization Tools:** Created interactive visualizations of traffic flow, routes, and network behavior.
* **Interactive Modes:** Built-in user-triggered situations for in-real-time simulation testing and accommodation.

1. **Sandipta Debnath:**

**Project Report Leadership:**

* Organized and outlined the project proposal with clear-cut objectives, approaches, and intended effects.
* Influenced compliance of technical deployment with project vision by ensuring documentation.
* Improved stakeholder communication through proposal modification of preciseness, coherence, and persuasiveness.

**7**. **Conclusion:** This Project Implement a city traffic routing system using graph theory, where intersections are vertex and roads are weighted edges is traffic management. Using BFS & Dijkstra's algorithms to calculate the optimal path based on different cases. For instance, minimum intersections or minimum traffic. This project System handles traffic incident management, i.e., accidents and closures. Then Produces visualizations to display and path decisions.

**8.** **References:**

**[1]** [706aa99765db69de76e42b7e357f12c6.pdf](https://iasj.rdd.edu.iq/journals/uploads/2024/12/14/706aa99765db69de76e42b7e357f12c6.pdf)

**[2]** <https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm>

**[3]** <https://www.slideshare.net/slideshow/route-finder-using-bi-directional-bfs-dfs/274575168>