Route Optimization System

Mukul Jangid

CS 665 Project

Introduction

- **System Purpose:** Optimizes routing for applications like logistics and urban planning.
- **Key Objectives:**
  - Simplifies complex route calculations through visualization.
  - Offers a choice of algorithms for tailored accuracy and speed.
  - Ensures efficient processing with scalable architecture for growing demands.

Design Patterns Employed

1. Strategy
2. Builder
3. **Thread Pool**
4. Facade
5. **Singleton**

Strategy Pattern

- Allows interchangeable routing algorithms (**DijkstraStrategy**, **BellmanFordStrategy**).
- **Significance:**
  - Facilitates easy integration of new algorithms.
  - Decouples algorithm logic from user interface.

Strategy Pattern

```java
public interface RouteStrategy {
  List<Integer> calculateRoute(
      Graph<Integer, DefaultWeightedEdge> graph, int startVertex, int endVertex);
}
```

```java
public class DijkstraStrategy implements RouteStrategy {

  @Override
  public List<Integer> calculateRoute(
      Graph<Integer, DefaultWeightedEdge> graph, int startVertex, int endVertex) {
    DijkstraShortestPath<Integer, DefaultWeightedEdge> dijkstra = new DijkstraShortestPath<>(graph);
    return dijkstra.getPath(startVertex, endVertex).getVertexList();
  }
}
```

```java
public class BellmanFordStrategy implements RouteStrategy {

  @Override
  public List<Integer> calculateRoute(
      Graph<Integer, DefaultWeightedEdge> graph, int startVertex, int endVertex) {
    BellmanFordShortestPath<Integer, DefaultWeightedEdge> bellmanFord =
        new BellmanFordShortestPath<>(graph);
    return bellmanFord.getPath(startVertex, endVertex).getVertexList();
  }
}
```

Builder Pattern

- Constructs complex graph objects incrementally.
- Significance:
  - Enhances code readability and manageability.
  - Supports various graph configurations flexibly.

Builder Pattern

```java
public GraphBuilder<V, E> addEdge(V sourceVertex, V targetVertex, double weight) {
  graph.addVertex(sourceVertex);
  graph.addVertex(targetVertex);
  E edge = graph.addEdge(sourceVertex, targetVertex);
  graph.setEdgeWeight(edge, weight);
  return this;
}
```

```java
public GraphBuilder<V, E> addVertex(V vertex) {
  graph.addVertex(vertex);
  return this;
}
```

Thread Pool Pattern

- Manages concurrent execution of route calculations.
- **Significance:**
  - Improves performance by managing thread lifecycle efficiently.
  - Reduces system resource overhead.

Thread Pool Pattern

```java
public RouteCalculationExecutor(Graph<Integer, DefaultWeightedEdge> graph, int threadCount) {
  this.graph = graph;
  this.executor = Executors.newFixedThreadPool(threadCount);
}

public void executeRouteCalculation(int startVertex, int endVertex, RouteStrategy strategy) {
  executor.submit(
      () -> {
        try {
          List<Integer> route = strategy.calculateRoute(graph, startVertex, endVertex);
          if (route != null && !route.isEmpty()) {
            String formattedRoute = formatRoute(route);
            LOGGER.info(
                "Route from " + startVertex + " to " + endVertex + ":\n" + formattedRoute);
            appendRouteToFile(formattedRoute);
          } else {
            LOGGER.info("No available route from " + startVertex + " to " + endVertex);
          }
        } catch (Exception e) {
          LOGGER.log(Level.SEVERE, "Error calculating route", e);
        }
      });
}
```

Facade Pattern

- Simplifies interactions with complex subsystems through **RouteOptimizationFacade**.
- **Significance:**
  - Hides system complexity from end-users.
  - Provides a unified interface for easier maintenance and scalability.

Facade Pattern

```java
public RouteOptimizationFacade() {
  System.out.println("Loading road network...");
  try {
    BufferedReader reader = new BufferedReader(new FileReader(DATA_FILE_PATH));
    graph = RoadNetworkLoader.loadGraph(reader);
  } catch (IOException e) {
    System.err.println("Error loading graph: " + e.getMessage());
    graph = null;
  }
  if (graph != null) {
    System.out.println("Graph loaded successfully.");
    executor = new RouteCalculationExecutor(graph, 10);
  } else {
    System.out.println("Failed to load graph.");
  }
}

public void optimizeRoute(int startVertex, int endVertex, String algorithmName) {
  RouteStrategy strategy = getRouteStrategy(algorithmName);
  executor.executeRouteCalculation(startVertex, endVertex, strategy);
}
```

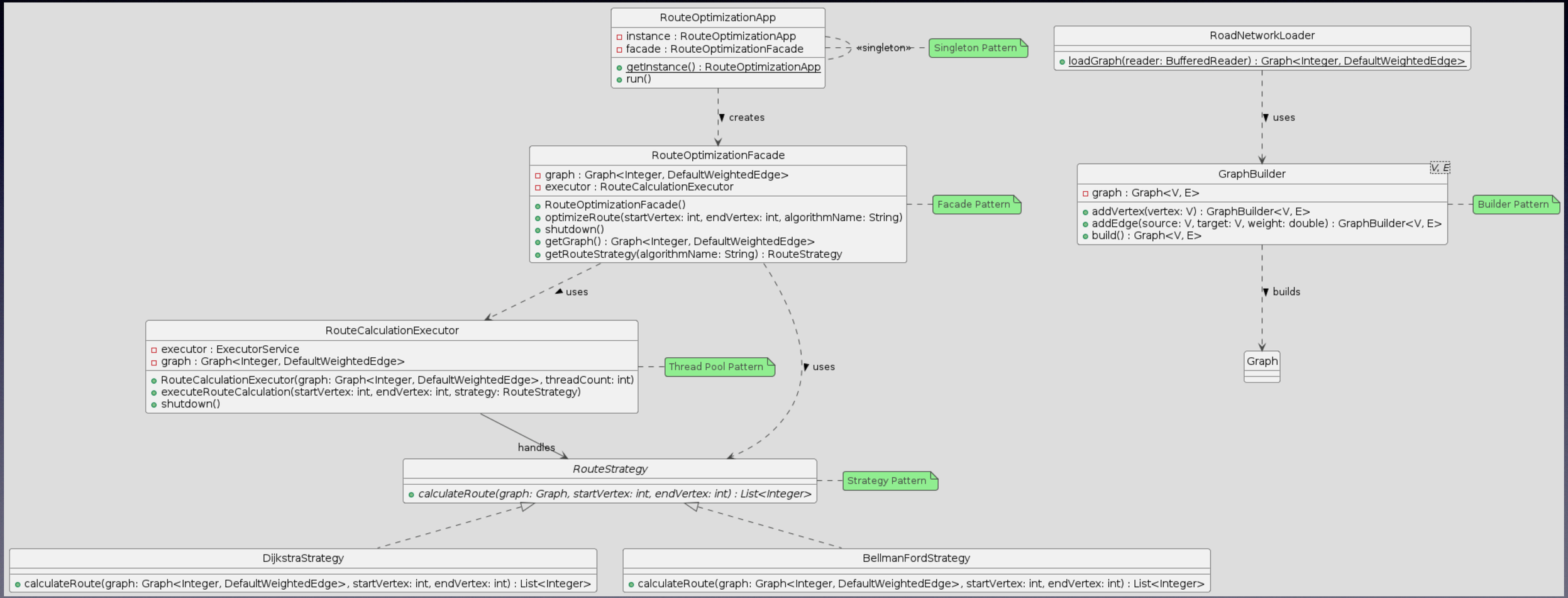
Singleton Pattern

- Ensures only one instance of the **RouteOptimizationApp**.

```java
public static synchronized RouteOptimizationApp getInstance() {
  if (instance == null) {
    instance = new RouteOptimizationApp();
  }
  return instance;
}
```

- **Significance:**
  - Manages application access.
  - Controls resources effectively.

System Architecture



Dataset Overview

- **California Road Network** [https://snap.stanford.edu/data/roadNet-CA.html]
- A road network of California. Intersections and endpoints are represented by nodes and the roads connecting these intersections or road endpoints are represented by undirected edges.

Dataset Overview

Dataset statistics	
Nodes	1965206
Edges	2766607
Nodes in largest WCC	1957027 (0.996)
Edges in largest WCC	2760388 (0.998)
Nodes in largest SCC	1957027 (0.996)
Edges in largest SCC	2760388 (0.998)
Average clustering coefficient	0.0464
Number of triangles	120676
Fraction of closed triangles	0.02097
Diameter (longest shortest path)	849
90-percentile effective diameter	5e+02

```
# Directed graph (each unordered pair of nodes is saved once): roadNet-CA.txt
# California road network
# Nodes: 1965206 Edges: 5533214
# FromNodeId    ToNodeId
0   1
0   2
0   469
1   0
1   6
1   385
2   0
2   3
```

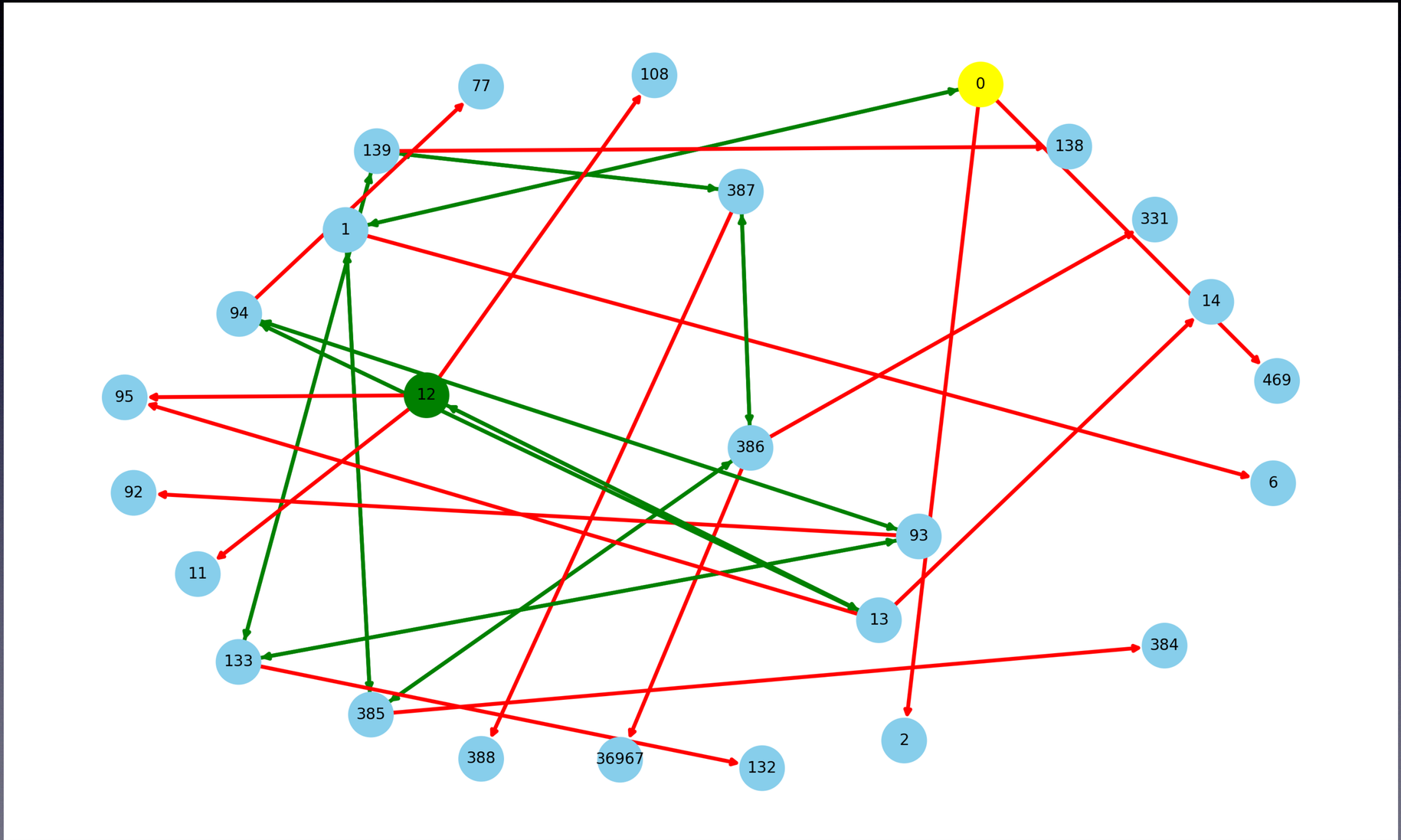
Vizualization with Python

- Enhances understanding and analysis by mapping computed routes and their complexities visually.
- Python Libraries Used:
  - **NetworkX:** For creating and manipulating complex network graphs.
  - **Matplotlib:** For plotting detailed visual representations of data.
- **Key Features of the Visualization:**
- **Subgraph Extraction:** Shows relevant portions of the graph to highlight specific routes and connections.
- **Color Coding:** Uses different colors to distinguish between main routes (green), additional connections (red), and key nodes (yellow for start, green for end).

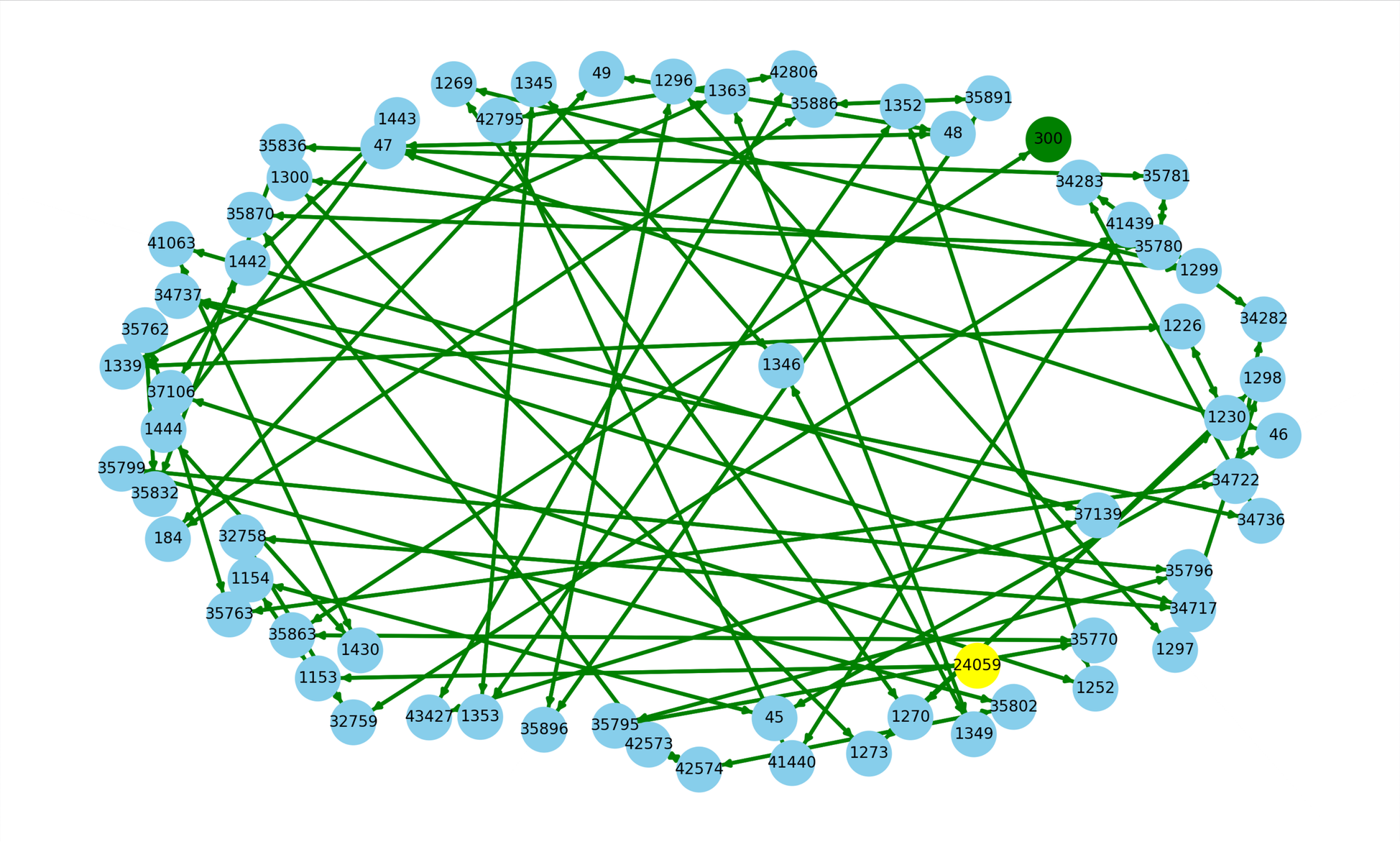
Output

```
Loading road network...
Processed 1000000 lines...
Processed 2000000 lines...
Processed 3000000 lines...
Processed 4000000 lines...
Processed 5000000 lines...
Graph loaded successfully.
Enter start and end vertices separated by space, choose algorithm: Dijkstra or BellmanFord ('exit' to quit)
0 12 Dijkstra
May 02, 2024 9:07:31 AM edu.bu.met.cs665.RouteCalculationExecutor lambda$executeRouteCalculation$0
INFO: Route from 0 to 12:
Path: 0 -> 1 -> 385 -> 386 -> 387 -> 139 -> 133 -> 93 -> 94 -> 13 -> 12
24059 300 BellmanFord
May 02, 2024 9:07:36 AM edu.bu.met.cs665.RouteCalculationExecutor lambda$executeRouteCalculation$0
INFO: Route from 24059 to 300:
Path: 24059 -> 1153 -> 1154 -> 45 -> 46 -> 47 -> 48 -> 49 -> 184 -> 35886 -> 35891 -> 35896 -> 1296 -> 1297 -> 1298 -> 1270 -> 1269 -> 1299 -> 1300
 -> 1273 -> 1230 -> 1226 -> 1339 -> 1363 -> 1349 -> 1346 -> 1345 -> 1353 -> 1352 -> 1252 -> 37106 -> 1442 -> 1443 -> 1444 -> 1430 -> 41063 -> 37139
 -> 43427 -> 42806 -> 42795 -> 41440 -> 41439 -> 32759 -> 32758 -> 34717 -> 34737 -> 34736 -> 34283 -> 34282 -> 34722 -> 35763 -> 35762 -> 35832 ->
 35836 -> 35781 -> 35780 -> 35870 -> 42573 -> 42574 -> 35802 -> 35799 -> 35796 -> 35795 -> 35770 -> 35863 -> 300
exit
```

Output


77
108
0
139
138
387
1
331
94
14
469
95
12
386
6
92
93
11
13
133
384
385
2
388
36967
132

Output


1269
1345
49
1296
42806
1363
35886
1352
35891
1443
42795
48
300
35836
47
1300
34283
35781
35870
41439
41063
35780
1442
1299
34737
34282
35762
1226
1339
1346
1298
37106
1230
1444
46
35799
34722
35832
37139
34736
184
32758
35796
1154
34717
35763
35863
1430
35770
1297
24059
1153
1252
35802
32759
43427
1353
35896
35795
45
1270
1349
42573
1273
42574
41440

Thank You!