A Tool for representing Traffic Congestion Mapping

Project Report submitted in partial fulfillment of the requirement

for the degree of

Bachelor of Technology

in

Computer Science & Engineering

under the Supervision of

Prof. Ravindara Bhatt

By

Kunal Chauhan (111250)

To

JAYPEE UNIVERSITY OF INFORMATION AND TECHNOLOGY WAKNAGHAT, SOLAN – 173234, HIMACHAL PRADESH, INDIA

Certificate

This is to certify that project report entitled “Traffic Congestion Mapping”, submitted by Kunal Chauhan in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science & Engineering to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Date:

Prof. Ravindara Bhatt
Assistant Professor

Acknowledgement

“The successful completion of any task would be incomplete without the support of the people who made it all possible and whose constant guidance and encouragement secured us the success.”

I feel proud and privileged in expressing my deep sense of gratitude to all those who have helped me in presenting this assignment. I would like to express my sincere gratitude to

Mr. Ravindara Bhatt for his inspiration, constructive suggestions, mastermind analysis and affectionate guidance in my work. It would have been impossible for me to complete this project without his guidance.

Lastly, I would like to add my deepest gratitude for the entire faculty of **Computer Science Department** of **Jaypee University of Information Technology** who helped me in learning the basics properly and implementing them accurately on my project.

Date: **Kunal Chauhan**

Table of Contents

S. No. Topic Page No

List of Figures

S. No. Title Page No

List of Tables

Abstract

Intelligent Transportation Systems (**ITS**) are advanced applications which, without embodying intelligence as such, aim to provide innovative services relating to different modes of transport and traffic management and enable various users to be better informed and make safer, more coordinated, and 'smarter' use of transport networks.
The project deals with **"Traffic Congestion Mapping"** which involves an approach to build efficient traffic management systems which would further help in reducing traffic problems and encourage safer transportation.

The framework used is a free, open-source software that provides a common and extendible understanding for the analysis and visualization of data that can be represented as a graph or network. Coding has been done on JAVA Platform and makes use of Graph Stream which is a Java Application Programming Interface (API) and includes libraries to support simulation of graphing algorithms. GraphStream is a graph handling Java library that focuses on the dynamics aspects of graphs. Its main focus is on the modeling of dynamic interaction networks of various sizes. The goal of the library is to provide a way to represent graphs and work on it.

The interface would use a map represented in the form of nodes and edges. The graph will therefore be used to generate traffic on the edges and finding all possible paths between the source and the end vertices. A portion of the graph can therefore be used to implement max flow problem by applying any of the algorithms and check for its optimal functioning on the portion of the map. The algorithm helps us to determine the augmenting paths along with the max flow generated along the start and end vertices.

1. INTRODUCTION

1.1 What is Intelligent Transportation System (ITS)?

A system where without actually using intelligence, we can aim to provide innovative services relating to different modes of transport and traffic management. It therefore enables various users to be better informed and make safer, more coordinated, and smarter use of transport networks. Fig 1.1 represents an ITS developed by Google to detect traffic and show it with different colors on the map so as to obtain paths with the least traffic [1].

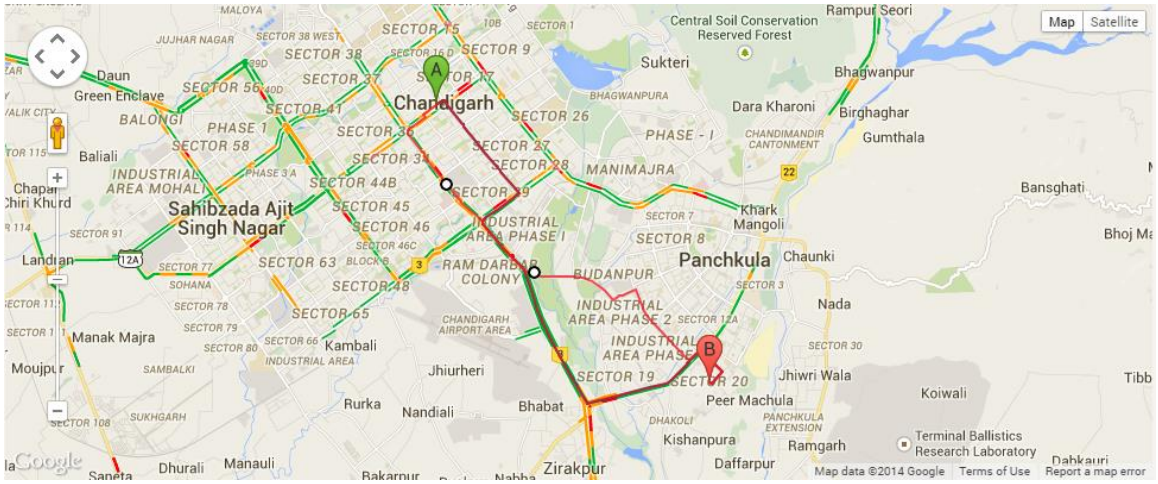

Fig 1.1 ITS developed by Google Traffic

A **traffic congestion map** is a graphical, real-time or near real-time representation of traffic flow for some particular area. Data is typically collected via anonymous GPS data points and loop sensors embedded in the roadways, then processed by computer at a central facility and distributed as a map view to users .

Many websites, news channels and mobile apps show these maps to help commuters avoid congested areas. Sometimes they are displayed directly to motorists using electronic signs. Frequently these show conditions on highways, but local streets can also be shown [12].

Similarly, the project Traffic Congestion Mapping involves the use of graph based techniques to represent road maps in the form of graphs and represent actual traffic on its

edges or use synthetic traffic generating algorithms to map that traffic. The nodes can be used to represent places on the graph while the edges can be used to represent the connecting routes to each node.

The weight associated with each edge would represent the corresponding distance on the roads or the graph. Traffic would be generated individually on each edge. Thereafter, we can use shortest-path calculating algorithms to obtain the short route from source to the destination nodes.

The path can therefore be highlighted to show the actual shortest path on the graph.

1.2 Problems/Challenges in ITS

- **ITS Design Requirements: Rugged, Small and Certified:** Transportation solutions are most often housed outdoors or in moving vehicles, where exposure to a variety of climates dictates the need to operate in extended temperatures and to support the extremes of shock, vibration and humidity. In addition, space restrictions require putting expanding functionality on ever-smaller board form factors.
- **Locomotive Data Video Recording (DVR) and Data Gateway:** A leading global supplier of technology solutions for railroads wanted to develop an onboard locomotive video/audio capture system to aid in accident investigations and provide safety training to crews. In addition to video and audio recording, requirements for the system included remote monitoring and control, real-time health monitoring and wireless video download.

  The video can be stored on hard disk drives (HDD) and can be accessed for any further use. The whole setup was too costly and complex which would not have been implemented effectively.
- In order to be acceptable to public and private interests ITS applications must be cost-effective, reliable, and easy to use and maintain. For public authorities, these systems must be able to show increases in efficiency, reductions in environmental

pollution, and most importantly reductions in the number of accidents involving injury and fatality.

- ITS applications that are perceived by the public to be hard to use or unsafe will not be acceptable. Ensuring adequate consumer information and, if necessary, training is also important although this must not exempt manufacturers from accepting responsibility for their products [12].

1.3 Applications of ITS

- **Emergency Vehicle Notification Systems**

  The in-vehicle eCall is an emergency call generated either manually by the vehicle occupants or automatically via activation of in-vehicle sensors after an accident. When activated, the in-vehicle eCall device will establish an emergency call carrying both voice and data directly to the nearest emergency point. The voice call enables the vehicle occupant to communicate with the trained eCall operator. At the same time, a minimum set of data will be sent to the eCall operator receiving the voice call.

- **Automatic Road Enforcement**

  A traffic enforcement camera system, consisting of a camera and a vehicle-monitoring device, is used to detect and identify vehicles disobeying a speed limit or some other road legal requirement and automatically ticket offenders based on the license plate number. Traffic tickets are sent by mail. Applications include:

1. Speed cameras that identify vehicles traveling over the legal speed limit. Many such devices use radar to detect a vehicle's speed or electromagnetic loops buried in each lane of the road.
2. Red light cameras that detect vehicles that cross a stop line or designated stopping place while a red traffic light is showing.

3. Bus lane cameras that identify vehicles traveling in lanes reserved for buses. In some jurisdictions, bus lanes can also be used by taxis or vehicles engaged in car pooling.

- **Variable Speed Limits**:

  Recently some jurisdictions have begun experimenting with variable speed limits that change with road congestion and other factors. Typically such speed limits only change to decline during poor conditions, rather than being improved in good ones.

- **Collision Avoidance Systems:**

  Japan has installed sensors on its highways to notify motorists that a car is stalled ahead [1][13].

1.4 Motivation

During the last few decades, the total number of vehicles around the world has grown. As a result, huge traffic has led to congestion on roads. Therefore, traffic being a real life problem tends to exist and needs to be managed. Based on the approach of ITS, several tools and utilities can be developed that can help in reducing the traffic congestion on roads as well as preventing some major accidents globally. The motivation for the project has been taken from the Intelligent Transportation Systems, using which an interface to represent road maps in terms of graphs can be made. The graphs can therefore be used to generate random traffic which can in the form of actual traffic representation models. Some of the types of traffic representation include Gaussian or Normal Traffic representation or Poisson Traffic representation. Here we will try to use Gaussian traffic as it is a popular approach toward traffic problem solving. Further, the shortest distance between the start and destination can be obtained by applying the shortest path algorithm. Shortest Path will be detected from the traffic data to guide the user through a path with least traffic. However, by expanding the interface we can use it to offer better visualization and develop an application based on it for better accessibility.

1.5 Objectives

Some of the main objectives covered by this project are:

- Representation of actual road maps in the form of graphs and studying them to solve traffic related problem.
- Simulation of synthetic traffic or random traffic by making use of traffic distribution models like Gaussian or Poisson distribution.
- Efficient implementation of some graph related algorithms and to study the optimization of those algorithms.
- Traffic generated can be used to detect congestion on the graph similar to detecting congestion on the road maps.
- This approach can be used to simulate traffic related to other cities by representing the road maps in the form of graphs.

1.6 Problem Statement

Developing a tool for Traffic Congestion Mapping to represent a map in the form a graph, making use of Normal or Gaussian distribution to generate traffic on each edge and using this congestion to show optimization by implementing max flow algorithm on a particular portion of the map.

The map of a planned city can be used to represent a graph that exactly represents the road structure of the actual map. The important stations can be represented as the nodes while the edges can be used to connect the nodes. The weight of the edges can be used to represent the approximate distance between two stations on the graph.

The Gaussian or Normal distribution can therefore be represented on the map along with each edge. The value of capacity i.e. the number of vehicles between each pair of nodes at a point of time is therefore determined by the Gaussian value. The capacity will also show the average number of vehicles occupying the road at a point of time.

After obtaining this value of capacity, we use a portion of this graph to implement Max Flow algorithm. The max flow algorithm that has been used here is the Ford-Fulkerson Algorithm which basically determines the flow between the source and the destination vertices entered by the user. It shows us whether there is any augmenting path associated with the graph or not i.e. if we can increase its capacity beyond a certain value of not. By implementing max flow on a small portion of the map of Chandigarh, we can get an optimal result and further use the algorithm on the whole map to check its optimality.

2. THEORY

The theory related to the use of some terms on this project has been described briefly for the proper understanding of the basics and to get proper knowledge of the project.

2.1 Graph Theory

Graphs (also known as networks) consist of a set of vertices, V, and a set of edges, E; the number of vertices is denoted by |V | and the number of edges by |E|. Vertices (also known as nodes) represent entities, and edges (also known as arcs, links, or ties), which connect vertices, represent relationships or events which involve the entities that the vertices represent. The number of edges incident to a vertex is called the degree of that vertex. Graphs in which each edge has an associated numeric value (such as the number of co-authored papers) are called weighted or valued graphs. One common graph subtype is a k-partite graph (called a bipartite graph when k = 2), in which the vertices are partitioned into k disjoint subsets, and each edge connects vertices in distinct partitions.

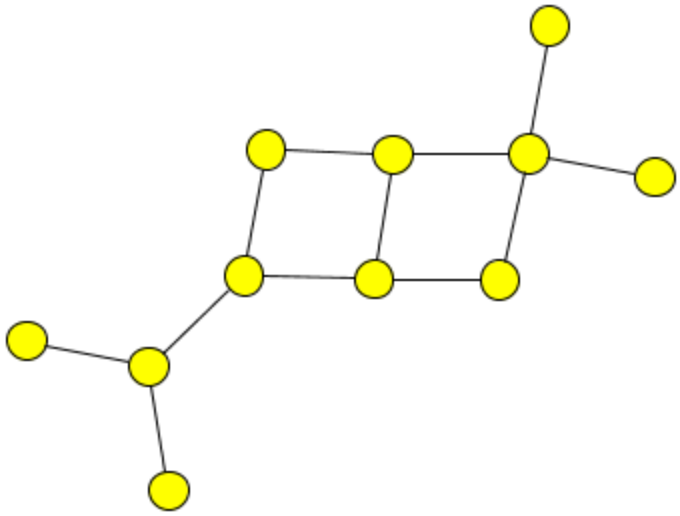

Fig 1.2 A Simple Graph

Most graphs contain edges that connects exactly two vertices; unless otherwise specified, all graphs in this paper have this property. (Graphs in which edges can connect any number of vertices are called hypergraphs, and their edges are called hyperedges.) An

edge which has a defined source and destination (such as one representing "A has cited B") are called directed edges; an edge which does not (such as one representing "A and B have co-authored a paper" are called undirected edges. Graphs which contain both directed and undirected edges are called mixed-type graphs. Two edges are said to be parallel if they connect the same set of vertices and have the same direction/ordering.

A graph is said to be connected if each vertex is reachable from each other vertex; many algorithms (such as centrality algorithms) are only well-defined on connected graphs. We define the distance between two vertices to be the length of the shortest path (on the underlying un-weighted graph) that connects them; in other contexts, the distance may refer to the shortest weighted path.

A network may contain entities of different types, or with different roles; it may also include different types of relationships or events. These roles and interaction types are collectively referred to as modes. A network which has one type of entity and one type of relationship is called a single-modal network; if the network has more than one type of entity, relationship, or both, it is called a multi-modal network [4][5].

A complex connection of nodes to edges can be used to obtain a large graph which in turn could represent a road map. The Fig 1.3 shows the graph representation of a road map.

The edges in the Fig 1.3 can be seen connecting the nodes which form an interconnecting network representing a road map. The weights can be seen showing the distance of the edges.

As it is important to maintain the topology of the graph and it is difficult to represent circular edges with planar graphs, in this project we will be using the map of a planned city. The city would be represented graphically with the help of nodes and edges. The first part of the project involves building a graph out a map of a planned city.

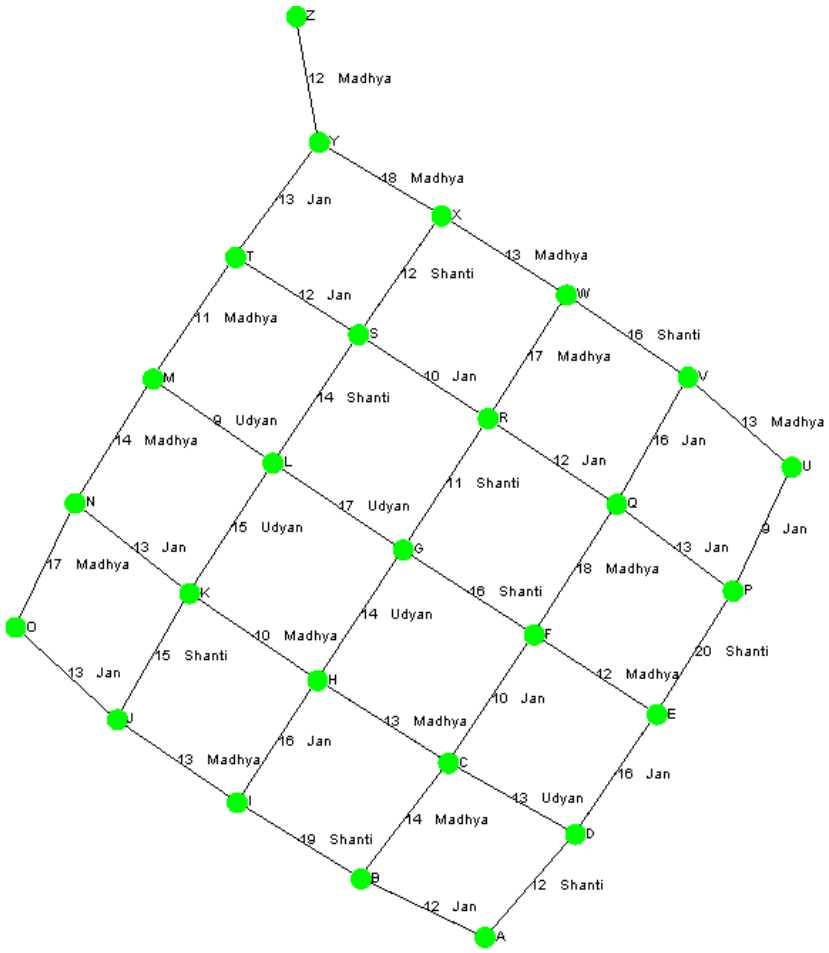

Fig 1.3 Graph Representation of a Road Map

2.2 Java and Object-Oriented Programming

Java is an object-oriented programming language. This generally means that programming involves:

(a) Defining object types and their capabilities, and

(b) Constructing objects and using their capabilities in aid of the desired tasks.

Object types are defined by interfaces and classes (which may implement one or more inter-faces, and may extend (inherit behavior from, or be a subclass of) a single other class). A particular object is said to be an instance of the types that it implements and

extends. Java defines a class called Object which is a superclass of all Java classes (including, implicitly, any user-defined class). The specific behaviors and capabilities of a class are defined by the implementation of that class's methods; the types and ordering of a method's arguments define that method's signature. Classes may be declared to be abstract, in which case they need not supply implementations of each method that they declare, and an instance of the class cannot be created; this can be useful for providing implementations that are general enough to apply to most anticipated extensions of such a class. Objects may contain references to other objects.

API (Application Programming Interface) is a common term for a software library, especially one whose design philosophy emphasizes backwards compatibility as the library evolves. The standard Java libraries, GraphStream and JUNG are APIs.

There are a few different Java APIs that can be used to create graphic user interfaces; two of the most popular are Swing (Sun Microsystems (2004)) and SWT (Eclipse Foundation (2001)).

2.3 Max-Flow Problem [9][10]

The maximum flow problem can be seen as a special case of more complex network flow problems, such as the circulation problem.

Definition:

Let $N=(V,E)$ be a network with $s,t \in V$ being the source and the sink of N respectively.

The **capacity** of an edge is a mapping $c: E \to \mathbb{R}^+$, denoted by c_{uv} or $c(u,v)$. It represents the maximum amount of flow that can pass through an edge.

A **flow** is a mapping $f: E \to \mathbb{R}^+$, denoted by f_{uv} or $f(u,v)$, subject to the following two constraints:

1. $f_{uv} \leq c_{uv}$, for each $(u,v) \in E$ (capacity constraint: the flow of an edge cannot exceed its capacity)
2. $\sum_{u:(u,v)\in E} f_{uv} = \sum_{u:(v,u)\in E} f_{vu}$, for each $v \in V \setminus \{s,t\}$ (conservation of flows: the sum of the flows entering a node must equal the sum of the flows exiting a node, except for the source and the sink nodes).

The **value of flow** is defined by $|f| = \sum_{v:(s,v)\in E} f_{sv}$, where s is the source of N. It represents the amount of flow passing from the source to the sink.

The **maximum flow problem** is to maximize $|f|$, that is, to route as much flow as possible from s to t.

Figure 1.4 shows a flow network, with source s and sink t. The numbers next to the edge are the capacities.

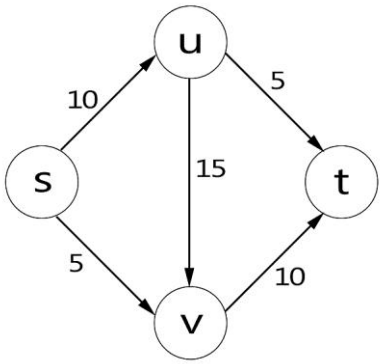

Fig 1.4 A graph showing edges with capacities.

Solutions

We can define the **Residual Graph**, which provides a systematic way to search for forward-backward operations in order to find the maximum flow.

Given a flow network G, and a flow f on G, we define the residual graph G_f of G with respect to f as follows.

1. The node set of G_f is the same as that of G.

2. Each edge $e=(u,v)$ of G_f is with a capacity of $c_e - f(e)$.

3. Each edge $e'=(v,u)$ of G_f is with a capacity of $f(e)$.

Given below are a few techniques, along with their complexities to solve the max flow problem:

Method	Complexity	Description
Linear programming		Constraints given by the definition of a legal flow.
Ford–Fulkerson algorithm	$O(E \max\|f\|)$	As long as there is an open path through the residual graph, send the minimum of the residual capacities on the path. The algorithm works only if all weights are integers. Otherwise it is possible that the Ford–Fulkerson

		algorithm will not converge to the maximum value.
Edmonds–Karp algorithm	$O(VE^2)$	A specialization of Ford–Fulkerson, finding augmenting paths with breadth-first search.
Dinic's blocking flow algorithm	$O(V^2E)$	In each phase the algorithms builds a layered graph with breadth-first search on the residual graph. The maximum flow in a layered graph can be calculated in $O(VE)$ time, and the maximum number of the phases is n-1. In networks with unit capacities, Dinic's algorithm terminates in $O(E\sqrt{V})$ time.
General push-relabel maximum flow algorithm	$O(V^2E)$	The push relabel algorithm maintains a preflow, i.e. a flow function with the possibility of excess in the vertices. The algorithm runs while there is a

		vertex with positive excess, i.e. an active vertex in the graph. The push operation increases the flow on a residual edge, and a height function on the vertices controls which residual edges can be pushed. The height function is changed with a relabel operation. The proper definitions of these operations guarantee that the resulting flow function is a maximum flow.
Push-relabel algorithmwith*FIFO* vertex selection rule	$O(V^3)$	Push-relabel algorithm variant which always selects the most recently active vertex, and performs push operations until the excess is positive or there are admissible residual edges from this vertex.
Dinic's algorithm	$O(VE \log(V))$	The dynamic trees data structure speeds up the maximum flow computation in the layered

		graph to $O(E\log(V))$.
Push-relabel algorithm with dynamic trees	$O(VE \log(V^2/E))$	The algorithm builds limited size trees on the residual graph regarding to height function. These trees provide multilevel push operations.
Binary blocking flow algorithm[8]	$O(E \min(V^{2/3}, \sqrt{E}) \log(V^2/E) \log U)$	The value U corresponds to the maximum capacity of the network.

Table 1.1 Solutions to solve Max-Flow Problem

However, some of these techniques are complex to implement. So, we shall mainly focus on Ford-Fulkerson Algorithm to detect the max flow.

3. PROPOSED WORK

The main requirement of the project is representation of traffic on a graph that represents the actual map of a city. This means that we need to model a map by converting it into a graph. After we have the graph, we can use its edges to represent the actual network of roads connecting the nodes. The traffic shall be generated using one of the random traffic generation methods mainly by Gaussian distribution thereby assigning each edge a random amount of traffic. Lastly, after having the traffic and distance values, the user shall enter the source and destination points looking at the graph. Thereafter, shortest path algorithm mainly Depth-First Search will be used to find all possible paths between the source and destination vertex. Next, we are going to focus on optimization by implementing the Max-flow problem mainly by using the Ford-Fulkerson Algorithm. Here, we shall be taking a portion of the map and using it to detect the augmenting paths as well as the flow between the source and the destination vertices. This will help us prove that the project works optimally by implementing the Ford-Fulkerson Algorithm.

In first part of the project main emphasis has been laid on map construction. The random traffic generation and implementation of Max-Flow Problem will be covered in the next part of the project. Below in the figure 1.5 (a) there can be seen a highlight of the work done on the project where I have successfully been able to construct the map and find out all possible paths between the source and the destination vertices.

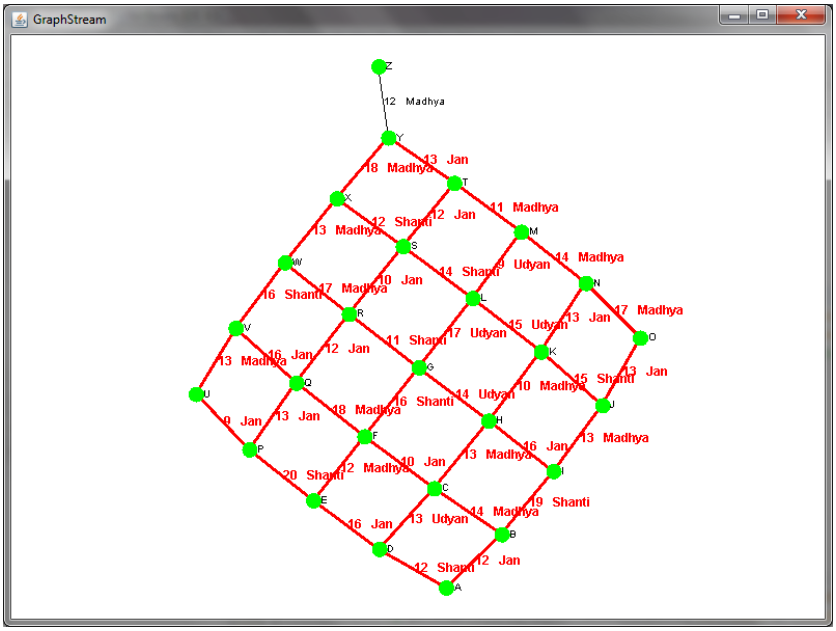

Fig 1.5 (a) The Actual Project Outlook

For random traffic generation we have used Gaussian distribution which generates Gaussian values of traffic on each edge at each execution. Further, these traffic values can be used to calculate congestion on each edge, which will help us implement the max flow problem to check for optimization. In figure1.5 (b) we can a portion of the map which can be used to implement max flow algorithm.

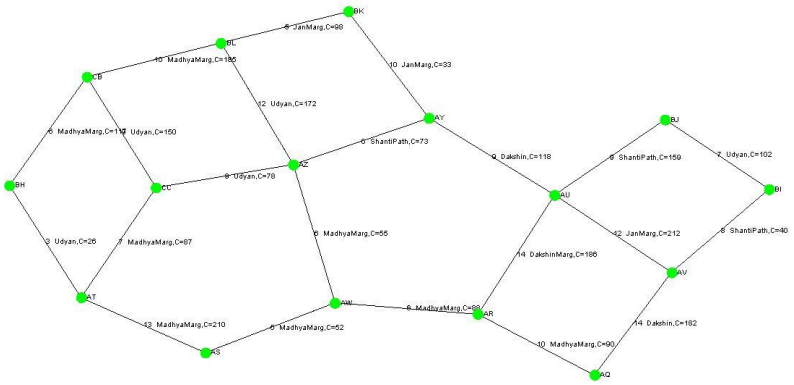

Fig 1.5 (b) Sample graph that can be used to implement Max Flow

Given below is the flowchart to demonstrate the proposed work.

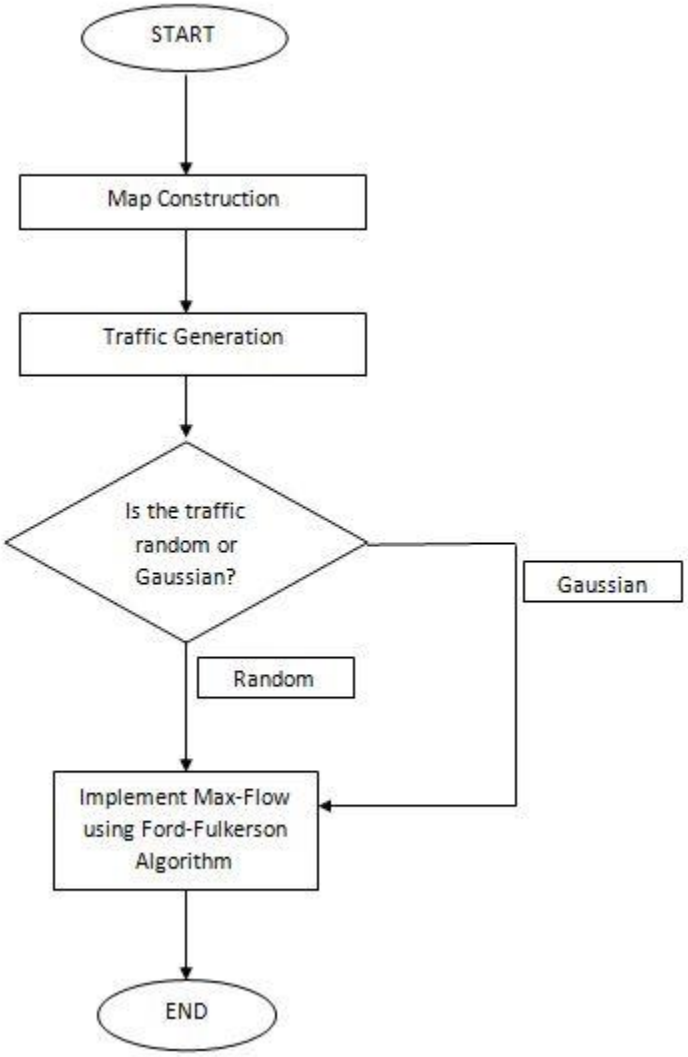

Fig 1.6 Flowchart to demonstrate the proposed work

3.1 Simulation

For simulating the graph, **GraphStream** has been used. GraphStream is a graph handling Java library that focuses on the dynamics aspects of graphs. Its main focus is on the modeling of dynamic interaction networks of various sizes.

The goal of the library is to provide a way to represent graphs and work on it. To this end, GraphStream proposes several graph classes that allow to model directed and undirected graphs, 1-graphs or p-graphs (a.k.a. multigraphs, that are graphs that can have several edges between two nodes).

GraphStream allows storing any kind of data attribute on the graph elements: numbers, strings, or any object.

Moreover, in addition, GraphStream provides a way to handle the graph evolution in time. This means handling the way nodes and edges are added and removed, and the way data attributes may appear, disappear and evolve.

In order to handle dynamic graphs, the library defines in addition to graph structures the notion of "stream of graph events", which as you guessed, is at the origin of the library name. The number of events is restricted they are:

- Node addition,
- Node removal,
- Edge addition,
- Edge removal,
- Graph/node/edge attributes addition,
- Graph/node/edge attributes change,
- Graph/node/edge attributes removal.
- Step

Inside the library, a lot of components can generate such streams of events. These components are called *sources*. Other components can receive these events and process them; they are in fact very comparable to listeners, a concept widely used in the Java world. We call such components *sinks*.

- When a component is able to both receive graph events (sink) and produce them (source) we call it a *pipe*. The graph structures in GraphStream are pipes. There are many kinds of pipes that can act as filter removing some events, or adding more events, or allowing crossing the network, or communicating between threads. Fig 1.5 shows the graph representations in GraphStream [4]**.**

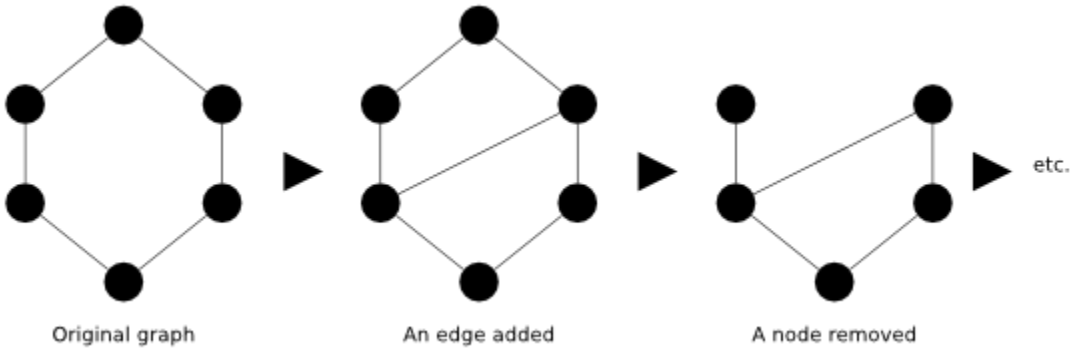

Fig 1.7 Graph Representations in GraphStream

3.1.1 Map Construction

Taking the map of planned city i.e. Chandigarh, I made a graph by looking at the actual map of Chandigarh and constructed the nodes which represented the main places of Chandigarh. The edges were constructed according to the roads joining the respective places marked as nodes. GraphStream is used to represent the actual graph which is the replica of the actual map of Chandigarh.

In order to construct the graph, GraphStream makes use of some libraries which include certain methods to construct the graph of desired size and complexity. The libraries are:

- **gs-core-1.2** It is a core library which contains the main methods than run in GraphStream.
- **gs-algo-1.2** It is used to implement graph algorithms on the constructed graphs.

- **gs-ui-1.2** It is used to associate styling and customization in the graph. It makes use of a **Cascading Style Sheet (CSS**) file to obtain the desired color and size of the nodes and the edges.

The graph representation of Chandigarh looks mostly like a graph in the form of a grid where all nodes can be reached through every other. Fig 1.8 shows the graph representation of Chandigarh city. From the figure it can be noticed that the edges have a label which represents the lane name while the weights are used to show the estimated distance.

The coordinates for the nodes are entered in a text file which is used as an input by the program. The text file contains the coordinates and the node names and the combination of nodes to create the edges along with the weight.

String Handling is used to split the string in the text file and obtain the coordinates as well as the nodes and edges.

3.1.2 Color and Size for Nodes and Edges (using Cascading Style Sheet)

The color as well as the size of the nodes and edges is defined in a **CSS** file that can be edited to use colors and font size of desired type. The **gs-ui-1.2** library contains methods that fetch the inputs from the CSS file and apply it on the graph. The CSS file contains classes to define the attributes of the graph to show different graphical representations after applying certain algorithms.

3.1.3 Classes, Methods and Functions

3.1.3.1 Classes

The program contains three classes namely:

1. **FileCoord:** This class contains methods to fetch the coordinates for the text file, apply string handling on the file, convert the string values to integer (if required) and plot the graph using the defined coordinates. It also obtains the input from the user and uses it in the methods to highlight all possible paths between two vertices. It also uses the **GraphGenerator** class construct the graph by adding every node and edge in the desired position and constructing the whole graph to represent a map.
2. **GraphGenerator:** This class contains the methods to add each node and edge to the graph and build the connection between the vertices. It is also used to define a bi-directional relationship between two nodes and check whether each node is connected to the other by an edge.
3. **Search:** The search class implements the Depth-First Search Algorithm by applying it on the constructed graph and and reconstructing the path using the visited nodes after successful implementation of the algorithm. This class consists of methods to find out all possible paths between the start and the destination vertices.
4. **MaxFlowFordFulkerson:** This class contains the code for implementing Ford-Fulkerson Algorithm on the portion of the graph taking from the original graph of Chandigarh city. It contains methods to find out all augmenting paths between the source and the destination vertices of the specified graph. Further, it makes use of the Gaussian random capacities of the traffic values to detect max flow between the specified vertices. The functions of this class are accessed through objects in the main file. The node structure used to store the vertices is a Hashmap that maps

each key-value pair. Adjacency List is used to store the capacities of the edges and called in the Ford-Fulkerson Algorithm to detect max flow between the specified vertices.

3.1.3.2 Methods and Functions

The three classes contain the following methods which perform the following operations:

1. ***findAllPaths*** This method is used to find all the path between the source and the destination nodes by making use of the **Depth-First Search** Algorithm.
2. ***highlightPath*** This method fetches the styling attributes to be applied on the graph from the CSS file and applies it on the graph.
3. ***addNode*** This method adds each node to the graph at the defined coordinate.
4. ***addEdge*** This method adds each edge can connects it to the nodes.
5. ***addTwoWayVertex*** This method constructs a bidirectional relationship between two nodes to show the two way traffic lanes.
6. ***isConnected*** This method checks that every node is connected to each other with a defined edge.
7. ***DepthFirst*** This method is used to calculate all possible distances between the start and end vertex entered by the user.
8. ***constructPath*** This method constructs the path traversed using the Depth-First Search and highlights it.
9. ***FindPath*** This method makes use of the capacities and uses them to calculate the augmenting paths between the start and end vertices.

3.1.4 Algorithms Implemented

Some algorithms have been implemented to carry out searching and perform path traversal to and from nodes. The max flow problem is also solved using the Foed-Fulkerson Algorithm.

3.1.4.1 Depth-First Search (DFS) Algorithm

Depth-first search **(**DFS**)** [6] is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

An Example

For the following graph shown in Fig 1.6:

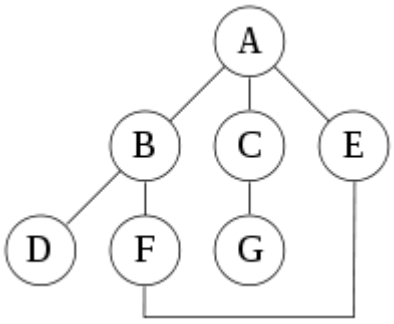

Fig 1.8 A graph to demonstrate DFS

A depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G. The edges traversed in this search form a Trémaux tree, a structure with important applications in graph theory.

Performing the same search without remembering previously visited nodes results in visiting nodes in the order A, B, D, F, E, A, B, D, F, E, etc. forever, caught in the A, B, D, F, E cycle and never reaching C or G.

Pseudocode

Input: A graph *G* and a vertex *v* of G

Output: All vertices reachable from *v* labeled as discovered

A recursive implementation of DFS:

```
1   procedure DFS (G,v):
2       label v as discovered
3       for all edges from v to w in G.adjacentEdges(v) do
4           if vertex w is not labeled as discovered then
5               recursively call DFS(G,w)
```

A non-recursive implementation of DFS:

```
1 procedure DFS-iterative(G,v):
2       let S be a stack
3       S.push(v)
4       while S is not empty
5           v ← S.pop()
```

6 **if** *v* is not labeled as discovered:

7 label *v* as discovered

8 **for all** edges from *v* to *w* **in** *G*.adjacentEdges(*v*) **do**

9 *S*.push(*w*)

These two variations of DFS visit the neighbors of each vertex in the opposite order from each other: the first neighbor of *v* visited by the recursive variation is the first one in the list of adjacent edges, while in the iterative variation the first visited neighbor is the last one in the list of adjacent edges. The **non-recursive** implementation is similar to **breadth-first search** but differs from it in two ways:

- It uses a stack instead of a queue, and
- It delays checking whether a vertex has been discovered until the vertex is popped from the stack rather than making this check before pushing the vertex.

3.1.4.2 Ford-Fulkerson Algorithm

The **Ford–Fulkerson method** or **Ford–Fulkerson algorithm** (**FFA**)[8][11]is an algorithm which computes the maximum flow in a flow network. It is called a "method" instead of an "algorithm" as the approach to finding augmenting paths in a residual graph is not fully specified or it is specified in several implementations with different running times It was published in 1956 by L. R. Ford, Jr. and D. R. Fulkerson The name "Ford–Fulkerson" is often also used for the Edmonds–Karp algorithm, which is a specialization of Ford–Fulkerson.

The idea behind the algorithm is as follows:

As long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of the paths. Then we

find another path, and so on. A path with available capacity is called an **augmenting path**.

Algorithm:

Let $G(V, E)$ be a graph, and for each edge from u to v, let $c(u, v)$ be the capacity and $f(u, v)$ be the flow. We want to find the maximum flow from the source s to the sink t. After every step in the algorithm the following is maintained:

Capacity constraints	$\forall (u,v) \in E \ f(u,v) \leq c(u,v)$	**The flow along an edge can not exceed its capacity.**
Skew symmetry**:**	$\forall (u,v) \in E \ f(u,v) = -f(v,u)$	The net flow from u to v must be the opposite of the net flow from v to u (see example).
Flow conservation**:**	$\forall u \in V : u \neq s \text{ and } u \neq t \Rightarrow \sum_{w \in V}$	That is, unless u is s or t. The net flow to a node is zero, except for the source, which "produces" flow, and the sink, which "consumes" flow.
Value(f)**:**	$\sum_{(s,u) \in E} f(s,u) = \sum_{(v,t) \in E} f(v,t)$	That is, the flow leaving from s must be equal to the flow arriving at t.

Table 1.2 Steps involved in Ford-Fulkerson Algorithm

This means that the flow through the network is a *legal flow* after each round in the algorithm. We define the **residual network** $G_f(V, E_f)$ to be the network with capacity $c_f(u, v) = c(u, v) - f(u, v)$ and no flow. Notice that it can happen that a flow from v to u is allowed in the residual network, though disallowed in the original network: if $f(u, v) > 0$ and $c(v, u) = 0$ then $c_f(v, u) = c(v, u) - f(v, u) = f(u, v) > 0$.

Algorithm - Ford–Fulkerson

Inputs Given a Network $G = (V, E)$ with flow capacity c, a source node s, and a sink node t

Output Compute a flow f from s to t of maximum value

- $f(u, v) \leftarrow 0$ for all edges (u, v)
- While there is a path p from s to t in G_f, such that $c_f(u, v) > 0$ for all edges $(u, v) \in p$:

1. Find $c_f(p) = \min\{c_f(u, v) : (u, v) \in p\}$
2. For each edge $(u, v) \in p$
   - $f(u, v) \leftarrow f(u, v) + c_f(p)$ (*Send flow along the path*)
   - $f(v, u) \leftarrow f(v, u) - c_f(p)$ (*The flow might be "returned" later*)

The path in step 2 can be found with for example a breadth-first search or a depth-first search in $G_f(V, E_f)$. If you use the former, the algorithm is called Edmonds–Karp.

When no more paths in step 2 can be found, s will not be able to reach t in the residual network. If S is the set of nodes reachable by s in the residual network, then the total capacity in the original network of edges from S to the remainder of V is on the one hand equal to the total flow we found from s to t, and on the other hand serves as an upper bound for all such flows. This proves that the flow we found is maximal. See also Max-flow Min-cut theorem.

If the graph $G(V, E)$ has multiple sources and sinks, we act as follows: Suppose that $T = \{t | t \text{ is a sink}\}$ and $S = \{s | s \text{ is a source}\}$. Add a new source s^* with an edge (s^*, s) from s^* to every node $s \in S$, with capacity $c(s^*, s) = d_s \ (d_s = \sum_{(s,u) \in E} c(s, u))$. And add a new sink t^* with an edge (t, t^*) from every node $t \in T$ to t^*, with capacity $c(t, t^*) = d_t \ (d_t = \sum_{(v,t) \in E} c(v, t))$. Then apply the Ford–Fulkerson algorithm.

Also, if a node u has capacity constraint d_u, we replace this node with two nodes u_{in}, u_{out}, and an edge (u_{in}, u_{out}), with capacity $c(u_{in}, u_{out}) = d_u$. Then apply the Ford–Fulkerson algorithm.

3.1.5 Traffic Generation

There are various ways of representing traffic. These days, satellites have sensors to monitor traffic by pointing out the geometric location of the vehicles and simulate them on the maps. But as we are using graphs to show similar kind of simulation, we will make use of random traffic to project traffic on the roadmap and use that traffic to find out congestion on the roads or edges. The main method of representing traffic is through Poisson distribution but here we are using Gaussian distribution to represent the traffic. Gaussian of Normal distribution is finding use in traffic representation and is used by many developers to simulate actual traffic data.

The traffic has been associated along each edge. Random class in Java makes it easy to use the random function. The random () function has been used to create an object which further has been used to access the value assigned to the random function parameter. RandomGaussian class has been used to create random Gaussian values associated with each edge. It contains the method getGaussian () that makes use of the mean and variance to calculate Gaussian values. The rand variable stores the randomly generated Gaussian values each time they are generated. Gaussian is also an object of class RandomGaussian that is used to access the method getGaussian () [7].

The assumption that has been made here is that each lane of the road i.e. left and right sides have vehicles between the range 1-100 at a point of time. The value keeps on changing at each execution depending on what the Gaussian value is generaed. Therefore, at each execution the value of the vehicles at each edge or lane is random and lies between 1-100.

On researching on the project, I found out that the average width of the highways of Chandigarh lies between **21.1 meters to 32.4 meters.** This value has been used to calculate the area of the road between each pair of node at each execution. The value consists the width of the main highways and the street lanes as not clearly shown on the map.

The method **r.nextDouble()** is used to generate each random value between the given ranges. The width of the lane can be calculated as given in the formula below:

Width= (r.nextDouble()*10) + 21.1

The area of the road between each pair of nodes can be calculated by the formula:

Area = length * width

This value of the area can further be multiplied with the random Gaussian value generated for the number of vehicles to obtain the amount of congestion on each lane at a point of time or the amount of road occupied by the vehicles. This also determines the average number of vehicles at a particular stretch of the road or on a particular edge.

The amount of congestion can be calculated by using the formula given below:

Congestion = (100 - rand) * ((int)e.getNumber("length") * width) / 100)

In the end we get the amount of area occupied by the vehicles. This value can also be expressed to show the lane with the least/most amount of congestion on the map which can be obtained by the formula given below:

Amount of congestion= 1 – 1/[(length*width)*(1/rand)]

In the end, both the values can be used to find the path which is the shortest for the user to follow.

The implementation of this part has been done in the Filecoord.java class. So, each time the program is executed the nodes and the edges are fetched from the text file and a new map is created with new values of vehicles between the lanes at a particular point of time.

So, this makes it as a real-life implementation of the actual representation of maps in the form of graphs and using those graphs to feed actual traffic data. This data, besides being represented on the map can be fed to various other applications to check their optimality using Max-Flow Algorithm.

Thereafter, viewing the congestion on the maps, the user can implement the max flow algorithm to check its optimal functioning on the portion of the graph.

3.1.6 Implementing Max-Flow Algorithm (Ford-Fulkerson Algorithm)

After generating traffic using Gaussian distribution, the original graph can be used to find out all possible paths based on the input entered by the user to choose the source and destination.

A portion of the graph has been used to implement the Max-Flow problem for which we have used the Ford-Fulkerson Algorithm to find out all the augmenting paths between the source and destination vertices and to check the graph for max flow between the two vertices. Given below is the graph on which the Ford-Fulkerson Algorithm has been implemented.

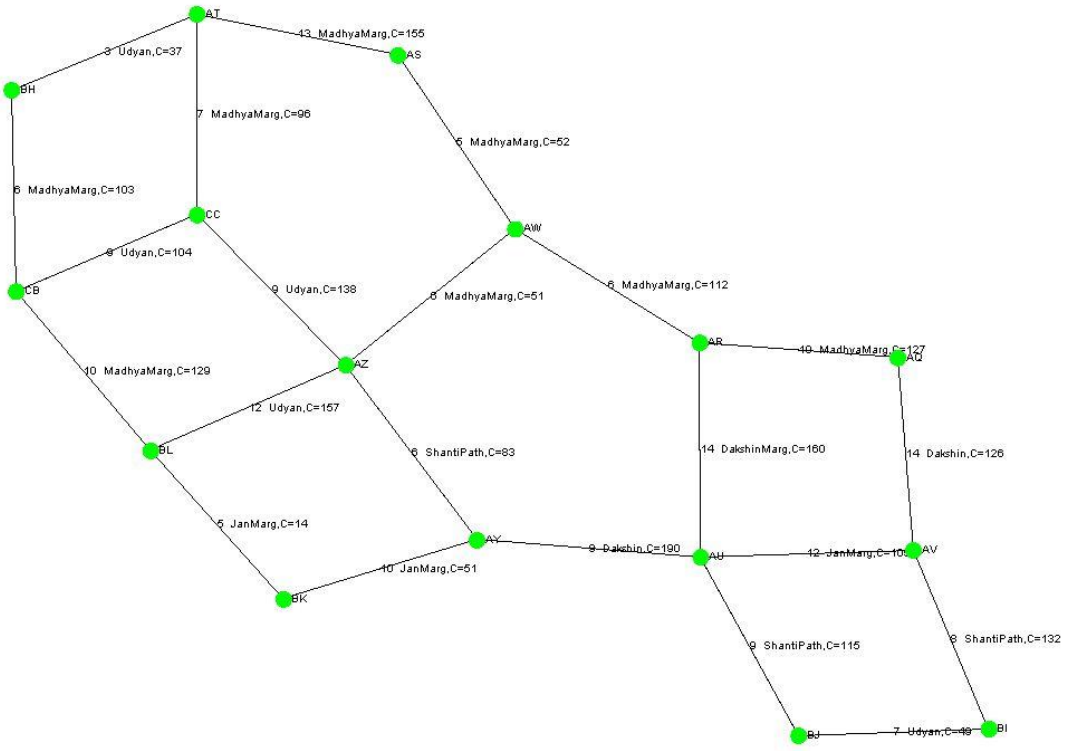

Fig 1.9 A portion from the actual map graph of Chandigarh city

The congestion variable stores the value of the average number of vehicles on each edge at a particular time which also represents the capacities associated with edge. This value is extracted and stored in an adjacency list from where the capacities are used to find out the augmenting path. These values are used in the Ford-Fulkerson Algorithm to calculate the max flow between the start and end vertices.

3.2 UML Diagrams

3.2.1 Class Diagram

A **Class Diagram** in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

In the diagram, classes are represented with boxes which contain three parts:

- The top part contains the name of the class. It is printed in bold and centered, and the first letter is capitalized.
- The middle part contains the attributes of the class. They are left-aligned and the first letter is lowercase.
- The bottom part contains the methods the class can execute. They are also left-aligned and the first letter is lowercase.

Fig 1.9 shows the class diagram to represent the traffic congestion mapping model.

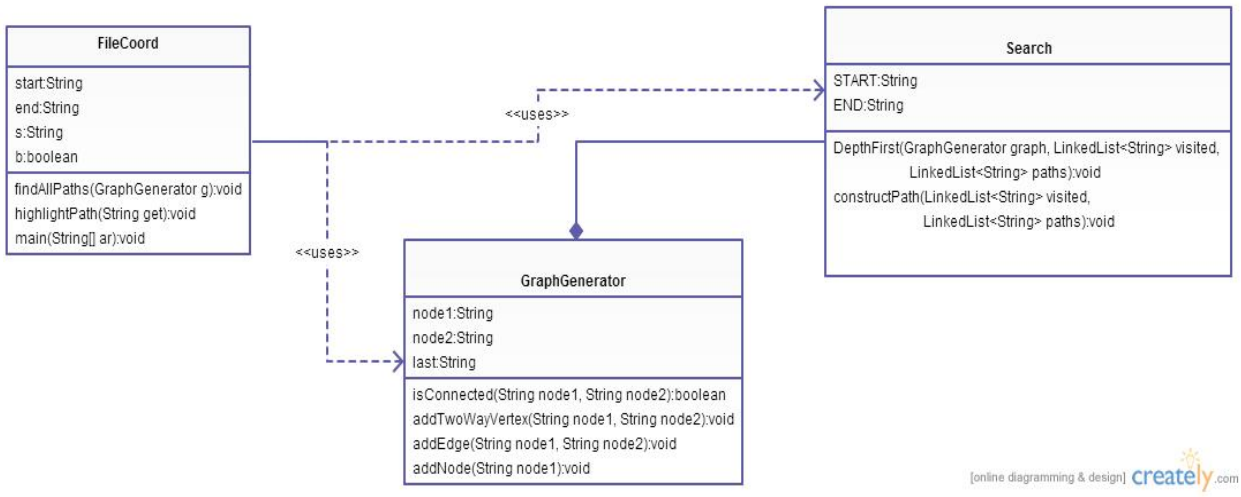

Fig 1.10 Class Diagram for Traffic Congestion Mapping

3.2.2 Use-Case Diagram

A **Use-Case Diagram** at its simplest is a representation of a user's interaction with the system and depicting the specifications of a use case. A use case diagram can portray the different types of users of a system and the case and will often be accompanied by other types of diagrams as well.

Fig 1.10 shows the use-case diagram with the actors and the system model for Traffic Congestion Mapping.

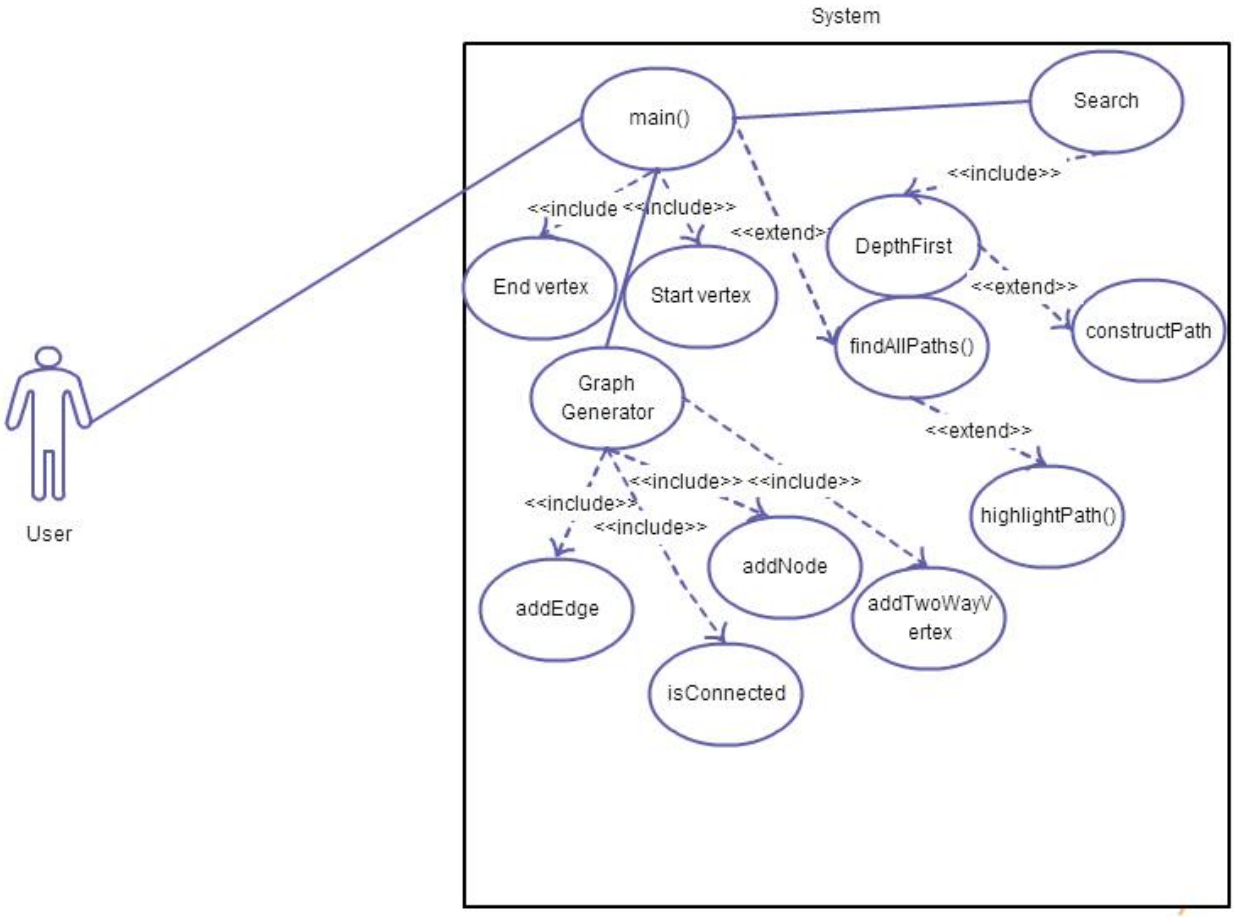

Fig 1.11 Use-Case Diagram for Traffic Congestion Mapping

5. OBSERVATIONS AND RESULTS

The text file coord.txt is used to get the information of the position of the coordinates and their connection using the edges. By making use of String Handling to separate the values of the coordinates as well as the nodes and edges, GraphStream constructs a graph of the city Chandigarh. Fig 1.11 shows the actual representation of the map in the form of a graph.

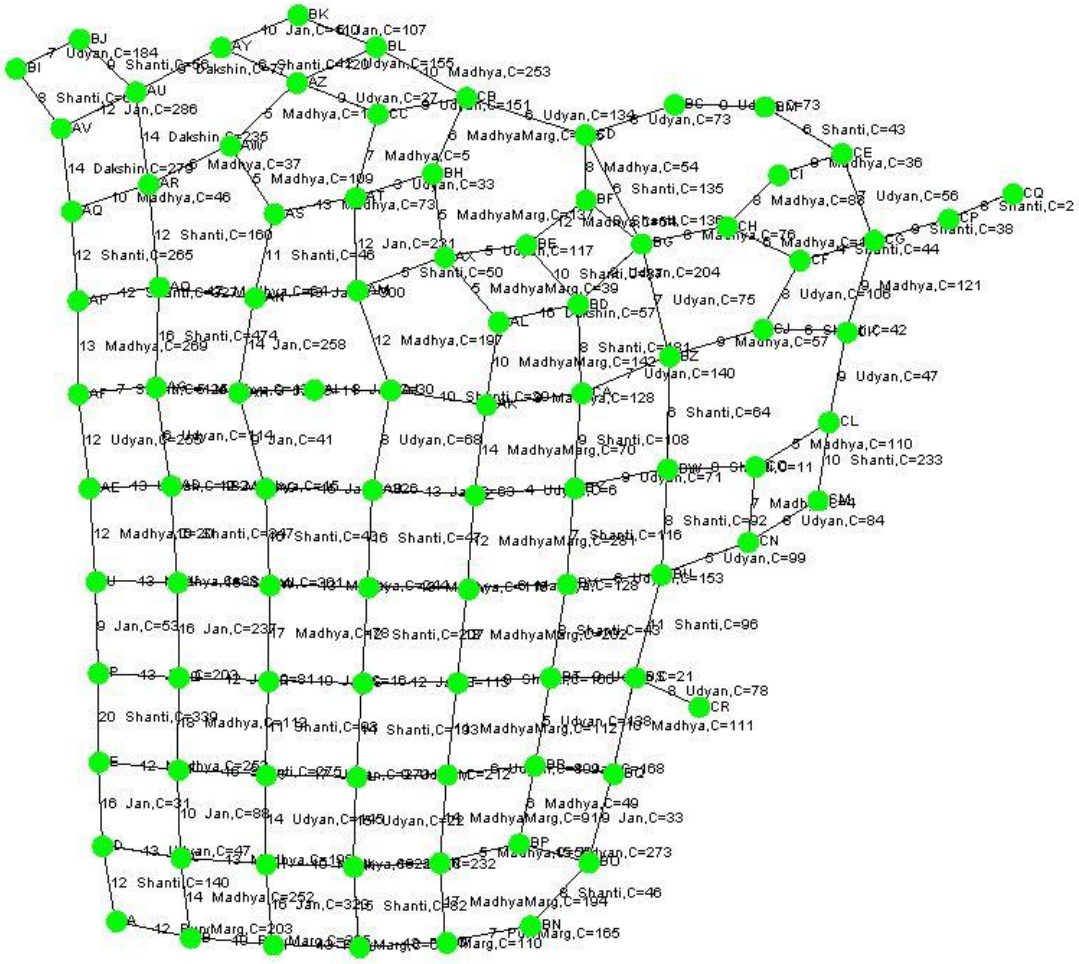

Fig 1.12 Graph to show the roadmap of Chandigarh

Given below is a sample text file that has been created for a portion of the map to apply the Ford-Fulkerson algorithm. Given below is the portion of the roadmap, on which the Ford-Fulkerson Algorithm has been implemented.

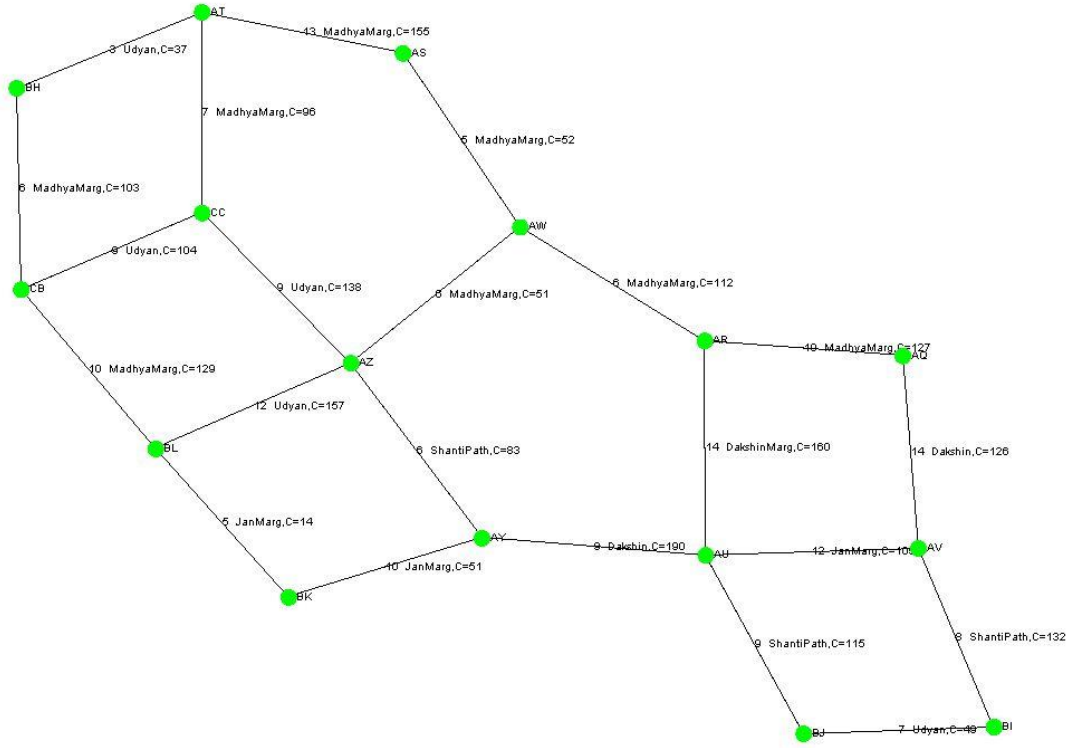

Fig 1.13 Portion of the Roadmap

Given below is the text file that contains the coordinates of the edges and the nodes which are used to define and display the map shown in Fig 1.12.The file Maxgraph.txt contains the nodes with the coordinates and again we have used string handling split the coordinates from the label [3].

Maxgraph.txt

- 2,6 AQ
- 1,4 AR
- 3,7 AS
- 2,4 AT
- 5,9 AU
- 6,9 AV

- 3,2 AW
- 3,8 AY
- 4,5 AZ
- 6,3 BH
- 5,7 BI
- 8,3 BJ
- 7,3 BK
- 7,4 BL
- 9,2 CB
- 7,5 CC
- Edges
- BH CB 6 MadhyaMarg
- CB CC 9 Udyan
- CC AT 7 MadhyaMarg
- AT BH 3 Udyan
- BL CB 10 MadhyaMarg
- BL AZ 12 Udyan
- AZ CC 9 Udyan
- BL BK 5 JanMarg
- BK AY 10 JanMarg
- AY AZ 6 ShantiPath
- AT AS 13 MadhyaMarg
- AS AW 5 MadhyaMarg
- AW AZ 6 MadhyaMarg
- AW AR 6 MadhyaMarg
- AR AU 14 DakshinMarg
- AU AY 9 Dakshin
- AU AV 12 JanMarg
- AV BI 8 ShantiPath
- BI BJ 7 Udyan

- BJ AU 9 ShantiPath
- AR AQ 10 MadhyaMarg
- AQ AV 14 Dakshin

Given below is the **incidence matrix** that shows the connection of the nodes with the edges.

	AQ	AR	AS	AT	AU	AV	AW	AY	AZ	BH	BI	BJ	BK	BL	CB	CC
AQ	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
AR	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
AS	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
AT	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
AU	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
AV	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
AW	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
AY	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
AZ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
BH	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
BI	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
BJ	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
BK	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
BL	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0
CB	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1
CC	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

Table 1.3 Incidence Matrix for the Roadmap

However, the value of the traffic generated corresponding to each edge is displayed when the program is executed. Given below the value of traffic generated along each edge. Note that the value of traffic generated is Gaussian in nature and has been rounded off to the nearest integer value. Also, the value of traffic is random and new on each execution.

- 0 BHCB (9,14) , Traffic = 34
- 1 CBBH (14,9) , Traffic = 34
- 2 CBCC (14,15) , Traffic = 75
- 3 CCCB (15,14) , Traffic = 75
- 4 CCAT (15,3) , Traffic = 66
- 5 ATCC (3,15) , Traffic = 66
- 6 ATBH (3,9) , Traffic = 69
- 7 BHAT (9,3) , Traffic = 69
- 8 BLCB (13,14) , Traffic = 56
- 9 CBBL (14,13) , Traffic = 56
- 10 BLAZ (13,8) , Traffic = 47
- 11 AZBL (8,13) , Traffic = 47
- 12 AZCC (8,15) , Traffic = 67
- 13 CCAZ (15,8) , Traffic = 67
- 14 BLBK (13,12) , Traffic = 42
- 15 BKBL (12,13) , Traffic = 42
- 16 BKAY (12,7) , Traffic = 50
- 17 AYBK (7,12) , Traffic = 50
- 18 AYAZ (7,8) , Traffic = 46
- 19 AZAY (8,7) , Traffic = 46
- 20 ATAS (3,2) , Traffic = 49
- 21 ASAT (2,3) , Traffic = 49
- 22 ASAW (2,6) , Traffic = 34
- 23 AWAS (6,2) , Traffic = 34
- 24 AWAZ (6,8) , Traffic = 38
- 25 AZAW (8,6) , Traffic = 38

- 26 AWAR (6,1) , Traffic = 59
- 27 ARAW (1,6) , Traffic = 59
- 28 ARAU (1,4) , Traffic = 45
- 29 AUAR (4,1) , Traffic = 45
- 30 AUAY (4,7) , Traffic = 58
- 31 AYAU (7,4) , Traffic = 58
- 32 AUAV (4,5) , Traffic = 48
- 33 AVAU (5,4) , Traffic = 48
- 34 AVBI (5,10) , Traffic = 81
- 35 BIAV (10,5) , Traffic = 81
- 36 BIBJ (10,11) , Traffic = 36
- 37 BJBI (11,10) , Traffic = 36
- 38 BJAU (11,4) , Traffic = 40
- 39 AUBJ (4,11) , Traffic = 40
- 40 ARAQ (1,0) , Traffic = 25
- 41 AQAR (0,1) , Traffic = 25
- 42 AQAV (0,5) , Traffic = 61
- 43 AVAQ (5,0) , Traffic = 61

The program is executed and the graph is formed by fetching the coordinates from the file specified above. The congestion parameters are computed and displayed along with the graph. The values of congestion are displayed in the output window as shown below:

- 0 BHCB (9,14) , Congestion = 83
- 1 CBBH (14,9) , Congestion = 83
- 2 CBCC (14,15) , Congestion = 53
- 3 CCCB (15,14) , Congestion = 53
- 4 CCAT (15,3) , Congestion = 69
- 5 ATCC (3,15) , Congestion = 69
- 6 ATBH (3,9) , Congestion = 27
- 7 BHAT (9,3) , Congestion = 27

- 8 BLCB (13,14) , Congestion = 105
- 9 CBBL (14,13) , Congestion = 105
- 10 BLAZ (13,8) , Congestion = 143
- 11 AZBL (8,13) , Congestion = 143
- 12 AZCC (8,15) , Congestion = 60
- 13 CCAZ (15,8) , Congestion = 60
- 14 BLBK (13,12) , Congestion = 87
- 15 BKBL (12,13) , Congestion = 87
- 16 BKAY (12,7) , Congestion = 126
- 17 AYBK (7,12) , Congestion = 126
- 18 AYAZ (7,8) , Congestion = 70
- 19 AZAY (8,7) , Congestion = 70
- 20 ATAS (3,2) , Congestion = 154
- 21 ASAT (2,3) , Congestion = 154
- 22 ASAW (2,6) , Congestion = 96
- 23 AWAS (6,2) , Congestion = 96
- 24 AWAZ (6,8) , Congestion = 82
- 25 AZAW (8,6) , Congestion = 82
- 26 AWAR (6,1) , Congestion = 68
- 27 ARAW (1,6) , Congestion = 68
- 28 ARAU (1,4) , Congestion = 191
- 29 AUAR (4,1) , Congestion = 191
- 30 AUAY (4,7) , Congestion = 89
- 31 AYAU (7,4) , Congestion = 89
- 32 AUAV (4,5) , Congestion = 184
- 33 AVAU (5,4) , Congestion = 184
- 34 AVBI (5,10) , Congestion = 39
- 35 BIAV (10,5) , Congestion = 39
- 36 BIBJ (10,11) , Congestion = 95
- 37 BJBI (11,10) , Congestion = 95
- 38 BJAU (11,4) , Congestion = 156

- 39 AUBJ (4,11) , Congestion = 156
- 40 ARAQ (1,0) , Congestion = 199
- 41 AQAR (0,1) , Congestion = 199
- 42 AQAV (0,5) , Congestion = 153
- 43 AVAQ (5,0) , Congestion = 153

However, the User has to enter the values of the source and the destination vertices by looking at the mapping of the nodes in the Hashmap. The output window displays the map –value associated with each node as shown below:

- AQ Map = 0
- AR Map = 1
- AS Map = 2
- AT Map = 3
- AU Map = 4
- AV Map = 5
- AW Map = 6
- AY Map = 7
- AZ Map = 8
- BH Map = 9
- BI Map = 10
- BJ Map = 11
- BK Map = 12
- BL Map = 13
- CB Map = 14
- CC Map = 15

In the end, after successful implementation of the algorithm, we find out all the augmenting paths between the source and the destination and also get the final max flow value. The augmenting paths can be used to practically solve the Ford-Fulkerson problem to check the correct functioning of the algorithm.

The augmenting paths obtained for the graph shown in figure 1.8 are:

- Flow increased By 127 **Augmenting path is 4->1->0**
- Flow increased By 105 **Augmenting path is 4->5->0**
- Flow increased By 21 **Augmenting path is 4->11->10->5->0**

However, for each execution the paths come out to be different due to the different values of capacities generated along each edge due to random Gaussian distribution.

The end result would show the final flow value as shown below:

- **Max Flow =253**

In the end we can infer that if the algorithm is working optimally for a portion of the graph, it would work optimally even by taking the whole graph of the city of Chandigarh thereby showing the correctness of the algorithm and its successful implementation on the project model.

Also, the comparison can be made by generating random traffic and then comparing it with the Gaussian values of traffic, which acts as a capacity for the max flow implementation. Given below is a graph to show the comparison between the random and Gaussian values of traffic.

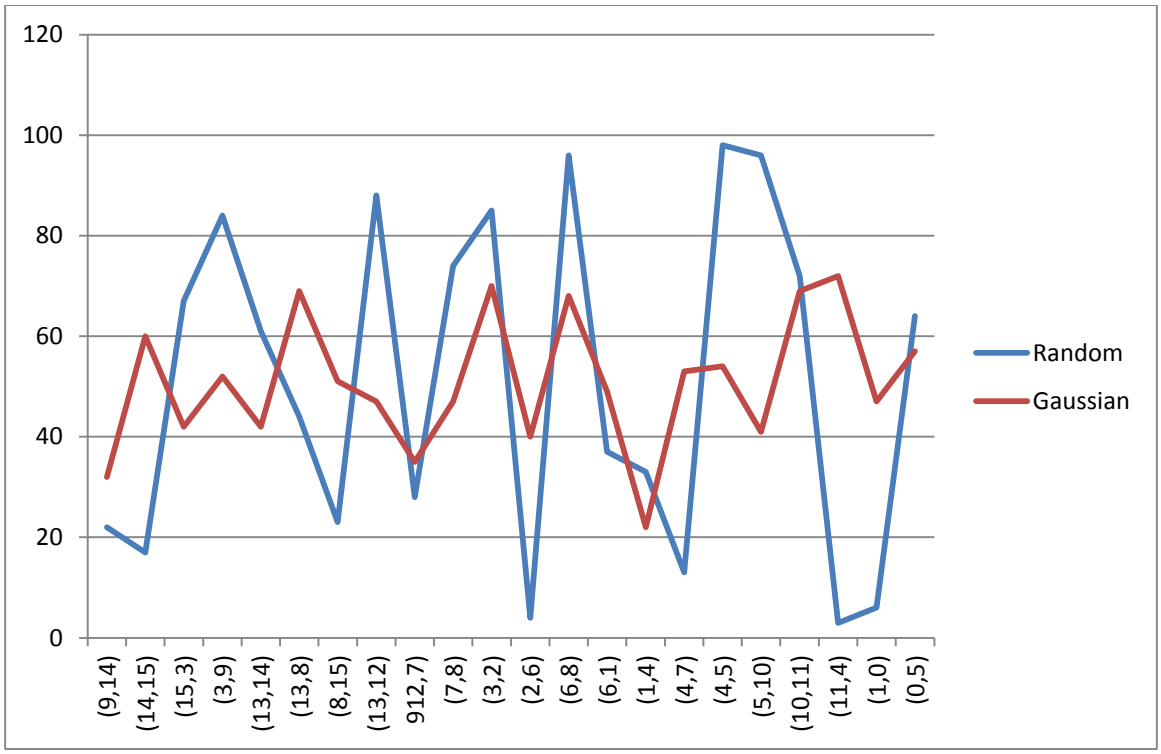

Fig 1.14 Graph to show the Random and Gaussian traffic comparison

4.1.2 User Input

On execution of the program, the max flow along with the augmenting paths and the mapped values of the nodes is displayed on the output screen. These mapped values can be used to determine the start and the end vertices as per the value of the node stored in the Hashmap. Each time the User wants to change the start and end vertices, he has to look for the specific value of the node on the Hashmap and enter it in the function.

In order to display all possible paths between the source and the destination, the User is prompted to enter the start and end vertices which are fed to the Depth-First Search Algorithm. Fig 1.12(a) and Fig 1.12(b) show the two dialog boxes where the User enters the input.

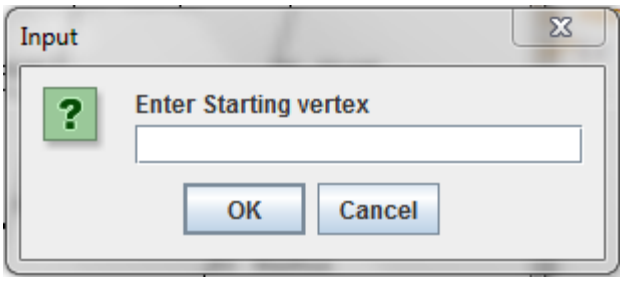

Fig 1.15(a) Dialog box to enter Start vertex

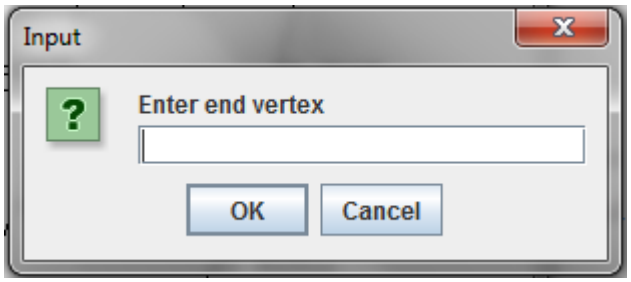

Fig 1.15(b) Dialog box to enter End vertex

After successful execution of the DFS algorithm we can see the all possible paths between the starting and the end vertex are highlighted. Fig 1.13 shows all possible paths between the nodes A and D highlighted with red color. The coloring has been shown using the Cascading Style Sheets as discussed above.

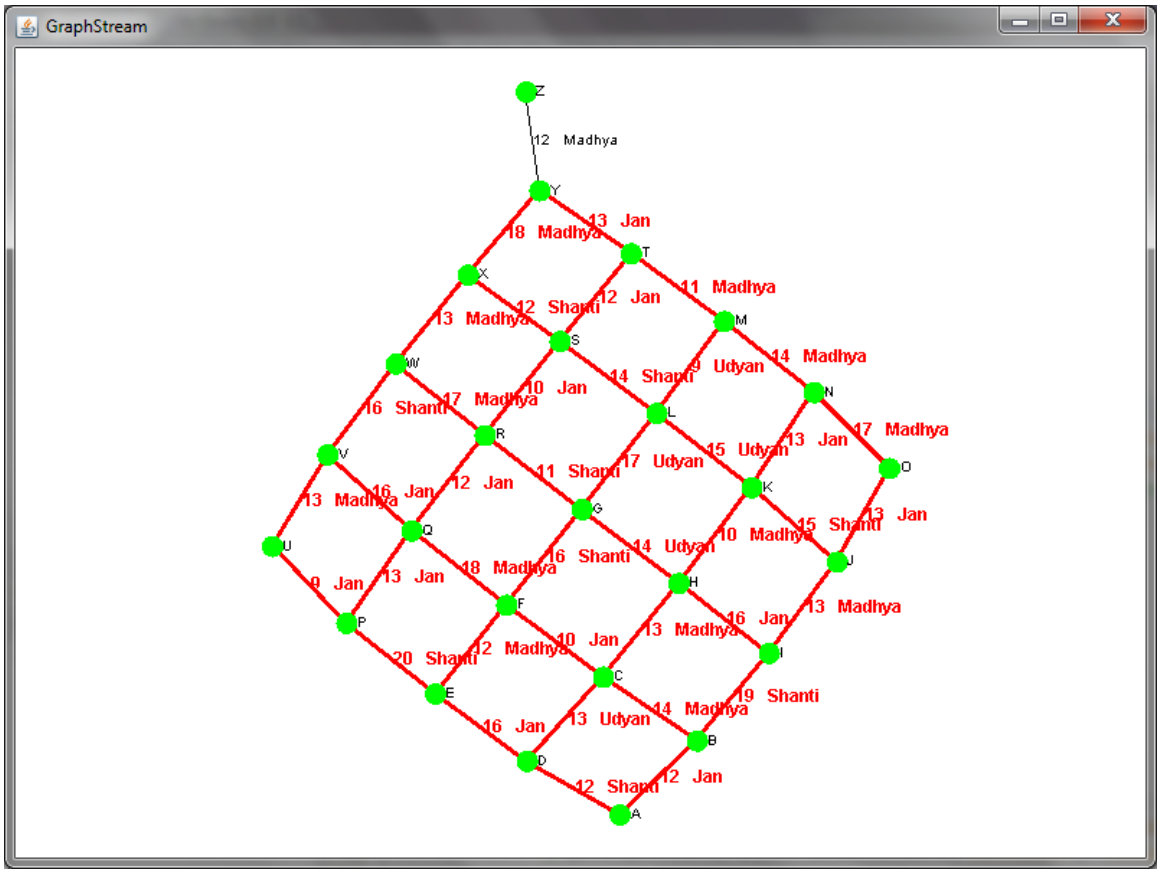

Fig 1.16 Graph showing all possible paths between A and D

Thus, as a result I was able to generate a graph of the map of the city Chandigarh and was successfully able to calculate all the possible paths between the source and destination vertices entered by the User.

Thereafter, I made use of Gaussian distribution to represent the traffic randomly on each edge and calculated congestion associated with each edge. The congestion value was further fed to the Max Flow function and used to detect all the augmenting paths and max flow between the entered vertices thereby highlighting successful implementation of the algorithm on the project.

6. CONCLUSION AND FUTURE WORK

Graphs can be used to represent road maps which can further be used to analyze and build traffic management systems. They can help in analyzing traffic conditions and choosing routes accordingly. In the form of an application, this can be used a tool to help users fight traffic conditions in daily life. Traffic being a real-time problem would tend to exist so, we need to develop tools to helps reduce heavy traffic conditions which would further lead to less accidents and coordinated traffic on every road.

The following project on Traffic Congestion Mapping can be expanded for:

- Developing Intelligent Transportation Systems that can efficiently manage traffic and help in avoiding traffic problems.
- The interface can be used to build a mobile application which would be a utility tool for the users which they can carry in their pockets and use in heavy traffic conditions.
- Integrating the module with better designing methods so as to provide a user-friendly interface. This will also help in providing more accuracy and dependability on the application or program.
- The congestion values combined with the distance of each edge, can be used to form a combination to find out the least distance and least traffic between the start and end points. Thus, helping people to get through heavy traffic easily and to avoid wastage of time.

7. REFERENCES

1. Grier N., Chabini L.," A new approach to compute minimum time path trees in FIFO time dependent networks in *Intelligent Transportation Systems",* Proceedings of the IEEE 5th International Conference,IEEE, 2002, 485-490.
2. Ben Alexander Wuest and Darka Mioc, "Visualization and modeling of traffic congestion in urban environments", 10th AGILE International Conference on Geographic Information Science,2007,10
3. William Kocay, Donald L. Krehe, "Paths and Walks", Graphs, Algorithms, and Optimization,2005,CRC Press,469.
4. http://graphstream-project.org/doc/Tutorials
5. http://en.wikipedia.org/wiki/Graph_theory
6. http://en.wikipedia.org/wiki/Depth-first_search
7. http://en.wikipedia.org/wiki/Normal_distribution
8. http://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm
9. http://graphstream-project.org/api/gs-algo/org/graphstream/algorithm/class-use/Algorithm.html#org.graphstream.algorithm.flow
10. http://graphstream-project.org/api/gs-algo/org/graphstream/algorithm/flow/FlowAlgorithm
11. http://graphstream-project.org/api/gs-algo/org/graphstream/algorithm/flow/FordFulkersonAlgorithm
12. en.wikipedia.org/wiki/Intelligent_transportation_system
13. Kashif Naseer Qureshi and Abdul Hanan Abdullah, "*A Survey on Intelligent Transportation Systems",* Middle-East Journal of Scientific Research 15 (5): 629-642, 2013