



# **TRAFFIC CONTROL SIMULATION**

Summer Project, 2010

Programming Club

Science and Technology Council

IIT Kanpur

**MENTOR : ANKESH KUMAR SINGH**

**BY : ATUL KUMAR SINHA**

**RAHUL MAJI**

**GAURUSH HIRANANDANI**

# 1 SIMULATION

## 1.1 What is simulation?

**Simulation** is the imitation of some real thing, state of affairs, or process. The act of simulating something generally entails representing certain key characteristics or behaviors of a selected physical or abstract system.

## 1.2 What are the benefits and usages?

Simulation is used in many contexts, such as simulation of technology for performance optimization, safety engineering, testing, training, education, and video games. Training simulators include flight simulators for training aircraft pilots in order to provide them with a life like experience. Simulation is also used for scientific modeling of natural systems or human systems in order to gain insight into their functioning. Simulation can be used to show the eventual real effects of alternative conditions and courses of action. Simulation is also used when the real system cannot be engaged, because it may not be accessible, or it may be dangerous or unacceptable to engage, or it is being designed but not yet built, or it may simply not exist.

## 1.3 What is a computer simulation?

A computer simulation (or "sim") is an attempt to model a real-life or hypothetical situation on a computer so that it can be studied to see how the system works. By changing variables, predictions may be made about the behavior of the system.

Computer simulation has become a useful part of modeling many natural systems in physics, chemistry and biology, and human systems in economics and social science (the computational sociology) as well as in engineering to gain insight into the operation of those systems. A good example of the usefulness of using computers to simulate can be found in the field of network traffic simulation. In such simulations, the model behavior will change each simulation according to the set of initial parameters assumed for the environment.

## 1.4 Examples :

### 1.4.1 City and Urban Simulation

### 1.4.2 Flight Simulation

### 1.4.3 Satellite Navigation Simulators

### 1.4.4 Traffic Simulation

## 2 TRAFFIC SIMULATION

### 2.1 What is Traffic Simulation?

Traffic simulation or the simulation of transportation systems is the mathematical modeling of transportation systems (e.g., freeway junctions, arterial routes, roundabouts, downtown grid systems, etc) through the application of computer software to better help plan, design and operate transportation systems.

Simulation of transportation systems started over forty years ago,[2] and is an important area of discipline in Traffic Engineering and Transportation Planning today. Various national and local transportation agencies, academic institutions and consulting firms use simulation to aid in their management of transportation networks.

### 2.2 Traffic as a Simulation Object

Road transportation, that is, efficient movement of people and goods through physical road and street networks is a fascinating problem. Traffic systems are characterized by a number of features that make them hard to analyze, control and optimize. The systems often cover wide physical areas, the number of active participants is high, the goals and objectives of the participants are not necessarily parallel with each other or with those of the system operator (system optimum vs. user optimum), and there are many system inputs that are outside the control of the operator and the participants (the weather conditions, the number of users, etc.).

In addition, road and street transportation systems are inherently dynamic in nature, that is, the number of units in the system varies according to the time, and with a considerable amount of randomness. The great number of active participants at present at the same time in the system means a great number of simultaneous interactions.

### 2.3 Applications

Traffic simulation models are useful from a microscopic, macroscopic and sometimes mesoscopic perspectives. Simulation can be applied to both transportation planning and to transportation design and operations. In transportation planning the simulation models evaluate the impacts of regional urban development patterns on the performance of the transportation infrastructure. Regional planning organizations use these models to evaluate what-if scenarios in the region, such as air quality to help develop land use policies that lead to more sustainable travel. On the other hand, modeling of transportation system operations and design focus on a smaller scale, such as a highway corridor and pinch-points. Lane types, signal timing and other traffic related questions are investigated to improve local system effectiveness and efficiency. While certain simulation models are specialized to model either operations or system planning, certain models have the capability to model both to some degree.

## 2.4 MICRO-SIMULATION

Road traffic micro-simulation models are computer models where the movements of individual vehicles travelling around road networks are determined by using simple car following, lane changing and gap acceptance rules. They are becoming increasingly popular for the evaluation and development of road traffic management and control systems.

By contrast, micro-simulation models provide a better, and 'purer', representation of actual driver behavior and network performance. They are the only modeling tools available with the capability to examine certain complex traffic problems (e.g. intelligent transportation systems, complex junctions, shockwaves, effects of incidents). In addition, there is the appeal to users of the powerful graphics offered by most software packages that show individual vehicles traversing across networks that include a variety of road categories and junction types. This visual representation of problem and solution in a format understandable to layman and professional alike can be a powerful way to gain more widespread acceptance of complex strategies.

## 3 PLATFORM USED

### 3.1 WHY JAVA?

- Though Java may not be as optimized as languages like C/C++, the in-built support for graphics and GUI development (AWT and Swing components), makes it preferable.
- Built-in methods provided by Java makes development very easy and efficient.
- It is very portable and platform independent.
- Java follows Object Oriented Programming (OOP) which is a very strong feature.

## 4 THE PROGRAM – TRAFFIC SIMULATOR

In this program we have tried to keep separate classes for every entity, so that it is easy to make any change to a particular entity without disturbing other parts of the program.

### 4.1 GUI Development

We used Java AWT and a bit of Swing Architecture to implement the GUI. The Abstract Window Toolkit (AWT) is Java's original platform-independent windowing, graphics, and user-interface widget toolkit. The AWT is now part of the Java Foundation Classes (JFC) — the standard API for providing a graphical user interface (GUI) for a Java program.

### 4.2 Terrain

As of now we have a simple and fixed terrain



With minor manipulations in the code it is very easy to construct new roads and bridges. Thus, in a way the program can work with a different terrain too. As an extension, one can implement an interface in which the user will provide information about the terrain so that it can be dynamically varied.

### 4.3 Making new Roads and Bridges

As of now, the program supports only horizontal and vertical roads. We tried to concentrate on the algorithms part rather than the visual appeal. A road takes six parameters in its constructor.  $(x1, y1)$  and  $(x2, y2)$ , the initial and final co-ordinates of the center of the road, (roadno) which acts like a unique identity for a particular road and the (level) which indicates the level of the bridge, for eg Level 0 means the road is a simple ground road, Level 1 means that the road is a Level 1 bridge.

### 4.4 Lanes

Since lane is defined as a simple path in which any vehicle can move in a particular direction only. In this version of the program, each road has two lanes allowing movement in complementary direction.

### 4.5 Junctions

For the terrain, the user has to construct the roads and bridges. The program automatically detects the junctions and its type. For eg. it can be a (+) type or from among the four kinds of (T) type. The program then automatically places the traffic lights accordingly at proper places and starts it.

## 4.6 Traffic Light System

The traffic light works in a cyclic manner. There can be three states of the traffic light – red (0), yellow (1) and green (2). A variable (called signal) associated with each traffic light is incremented by 1 in every 2 seconds. Based on the value of the variable the state (0, 1 or 2) of the traffic light was set.

For eg. For a (+) type junction, four traffic lights are accordingly placed and the following system is followed:

Signal	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
State	G	G	G	G	R	R	R	R	R	R	R	R	R	R	R	Y
State(No.)	2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	1

To take care of synchronization, the signal for each traffic light is **initialized** accordingly as follows:

1 <sup>st</sup> Traffic Light	Signal = 1
2 <sup>nd</sup> Traffic Light	Signal =13
3 <sup>rd</sup> Traffic Light	Signal = 9
4 <sup>th</sup> Traffic Light	Signal = 5

The value of the signal is reinitialized to 1 as soon as it reaches 15.

In a similar way traffic light at an (L) type junction which had three traffic lights was taken care of:

Signal	1	2	3	4	5	6	7	8	9	10	11	12
State	G	G	G	G	R	R	R	R	R	R	R	R
State(No.)	2	2	2	2	0	0	0	0	0	0	0	0

To take care of synchronization, the signal for each traffic light is **initialized** accordingly as follows :

1 <sup>st</sup> Traffic Light	Signal = 1
2 <sup>nd</sup> Traffic Light	Signal =9
3 <sup>rd</sup> Traffic Light	Signal = 5

The value of the signal is reinitialized to 1 as soon as it reaches 13.

## 4.7 Vehicles

Vehicles move with random speed.

### 4.7.1 Vehicle Addition:

Vehicles are added in the program in two ways :

(Note : We have assumed that vehicles could enter the terrain area from the outside world only. So (+) buttons(see fig.) are present only at those places where the road touches the boundary. The program automatically detects such roads and places the buttons. Each button has a unique identification number).

#### 4.7.1.1 Dynamically by the user during run-time

The user has two click two buttons one-by-one. The first button (see fig. 1) sets the initial position of the car and the 2<sup>nd</sup> and final click sets the final position.

#### 4.7.1.2 Random

By using the built in pseudo-random number generator function of Java, we created two random number generators. When the first random number is equal to the unique identification number of a button and the second random number is equal to the unique identification number of some other button, a car is initialized with initial position near the first button and final position near the second button.

#### 4.7.2 Collision Check

When any two vehicles in the same lane come very close to each other, the speed of the car which is behind is decreased. The cars rotate with constant speed so there is no point of collision during rotation. To take care of the fact that two cars should not be initialized simultaneously (read very close in time and at the same place) which is possible in case of random cars very often or in case when there is a red light at the immediate lane associated with a button and too many cars are already present in that lane so that it cannot accommodate any more vehicles, each vehicle is rendered outside the screen area. So a stack of vehicles is made outside the screen area and these vehicles start moving as soon as there is no vehicle in the screen region i.e. when the road is sufficiently clear.

#### 4.7.3 Vehicle Path

The path of the each vehicle is set as soon as it is initialized and updated at every junction according to the traffic density at that time. We have used Graph Theory to take care of the path. We will talk about Graph Theory soon.

### 4.8 Graph Theory

#### 4.8.1 What is a graph?

In mathematics and computer science, graph theory is the study of graphs: mathematical structures used to model pair wise relations between objects from a certain collection. A "graph" in this context refers to a collection of vertices or 'nodes' and a collection of edges that connect pairs of vertices. A graph may be undirected, meaning that there is no distinction between the two vertices associated with each edge, or its edges may be directed from one vertex to another.

#### Dijkstra's Algorithm

Dijkstra's algorithm, conceived by Dutch computer scientist Edsger Dijkstra in 1959, is a graph search algorithm that solves the single-source shortest path problem for a graph with

nonnegative edge path costs, producing a shortest path tree. This algorithm is often used in routing. An equivalent algorithm was developed by Edward F. Moore in 1957.

For a given source vertex in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. In our program junctions represent vertices and edges represents lanes. The weights of the edges are the distance between pair of junctions which lies on that lane and the traffic density on that lane.

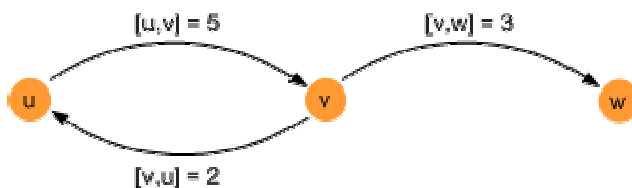
### Algorithm

Dijkstra's algorithm partitions vertices in two distinct sets, the set of *unsettled* vertices and the set of *settled* vertices. Initially all vertices are unsettled, and the algorithm ends once all vertices are in the settled set. A vertex is considered settled, and moved from the unsettled set to the settled set, once its shortest distance from the source has been found.

We all know that *algorithm + data structures = programs*, in the famous words of Niklaus Wirth. The following data structures are used for this algorithm:

- d stores the best estimate of the shortest distance from the source to each vertex
- $\pi$  stores the predecessor of each vertex on the shortest path from the source
- S the set of settled vertices, the vertices whose shortest distances from the source have been found
- Q the set of unsettled vertices

The distance between the vertex  $u$  and the vertex  $v$  is noted  $[u, v]$  and is always positive.



With those definitions in place, a high-level description of the algorithm is deceptively simple. With  $s$  as the source vertex:

```

// initialize d to infinity,  $\pi$  and Q to empty
d = (  $\infty$  )
 $\pi$  = ( )
S = Q = ( )
  
```

```

add s to Q
d(s) = 0

```

```

while Q is not empty
{
  u = extract-minimum(Q)
  add u to S
  relax-neighbors(u)
}

```

Dead simple isn't it? The two procedures called from the main loop are defined below:

```

relax-neighbors(u)
{
  for each vertex v adjacent to u, v not in S
  {
    if  $d(v) > d(u) + [u,v]$  // a shorter distance exists
    {
       $d(v) = d(u) + [u,v]$ 
       $\pi(v) = u$ 
      add v to Q
    }
  }
}

```

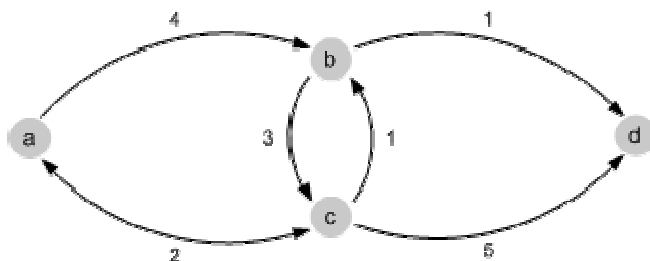
```

extract-minimum(Q)
{
  find the smallest (as defined by d) vertex in Q
  remove it from Q and return it
}

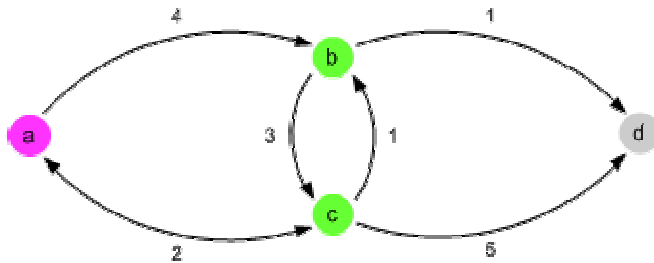
```

### An Example

So far I've listed the instructions that make up the algorithm. But to really understand it, let's follow the algorithm on an example. We shall run Dijkstra's shortest path algorithm on the following graph, starting at the source vertex *a*.



We start off by adding our source vertex  $a$  to the set  $Q$ .  $Q$  isn't empty, we extract its minimum,  $a$  again. We add  $a$  to  $S$ , then relax its neighbors. (I recommend you follow the algorithm in parallel with this explanation.)

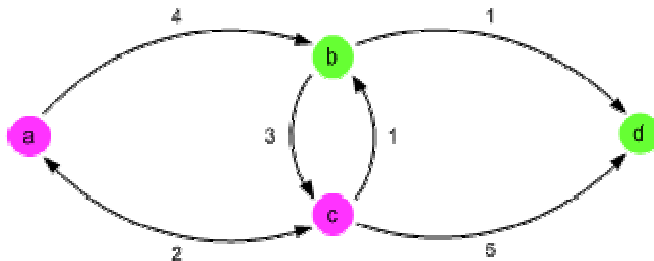


Those neighbors, vertices adjacent to  $a$ , are  $b$  and  $c$  (in green above). We first compute the best distance estimate from  $a$  to  $b$ .  $d(b)$  was initialized to infinity, therefore we do:

$$d(b) = d(a) + [a,b] = 0 + 4 = 4$$

$\pi(b)$  is set to  $a$ , and we add  $b$  to  $Q$ . Similarly for  $c$ , we assign  $d(c)$  to 2, and  $\pi(c)$  to  $a$ . Nothing tremendously exciting so far.

The second time around,  $Q$  contains  $b$  and  $c$ . As seen above,  $c$  is the vertex with the current shortest distance of 2. It is extracted from the queue and added to  $S$ , the set of settled nodes. We then relax the neighbors of  $c$ , which are  $b$ ,  $d$  and  $a$ .

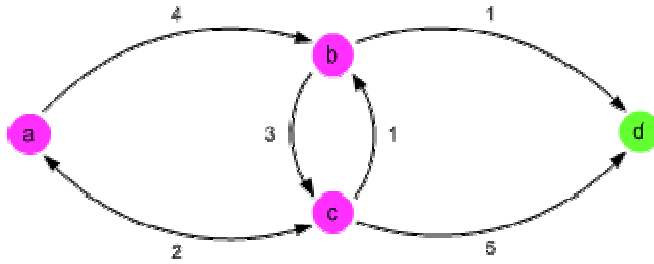


$a$  is ignored because it is found in the settled set. But it gets interesting: the first pass of the algorithm had concluded that the shortest path from  $a$  to  $b$  was direct. Looking at  $c$ 's neighbor  $b$ , we realize that:

$$d(b) = 4 > d(c) + [c,b] = 2 + 1 = 3$$

Ah-ah! We have found that a shorter path going through  $c$  exists between  $a$  and  $b$ .  $d(b)$  is updated to 3, and  $\pi(b)$  updated to  $c$ .  $b$  is added again to  $Q$ . The next adjacent vertex is  $d$ , which we haven't seen yet.  $d(d)$  is set to 7 and  $\pi(d)$  to  $c$ .

The unsettled vertex with the shortest distance is extracted from the queue, it is now  $b$ . We add it to the settled set and relax its neighbors  $c$  and  $d$ .



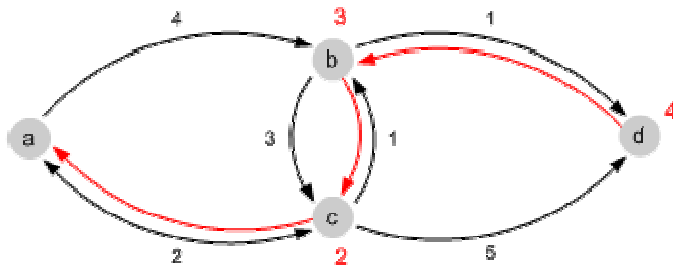
We skip  $c$ , it has already been settled. But a shorter path is found for  $d$ :

$$d(d) = 7 > d(b) + [b,d] = 3 + 1 = 4$$

Therefore we update  $d(d)$  to 4 and  $\pi(d)$  to  $b$ . We add  $d$  to the Q set.

At this point the only vertex left in the unsettled set is  $d$ , and all its neighbors are settled. The algorithm ends. The final results are displayed in red below:

- $\pi$  - the shortest path, in predecessor fashion
- $d$  - the shortest distance from the source for each vertex



This completes our description of Dijkstra's shortest path algorithm. Other shortest path algorithms exist (see the References section at the end of this article), but Dijkstra's is one of the simplest, while still offering good performance in most cases.

## 5 About the Classes

### 5.1 Car

*/\*Adds vehicles to the frame and simulates the same \*/*

**Car()**

*/\*The constructor takes four arguments: lane – initial lane number of the car; p[] – array that stores the sequential lane nos defining its the shortest and optimal path; junction[] – same as previous but stores junction nos instead of lane nos; finaljunction – the last element of junction[]\*/*

*/\*Acc. To lane nos and flag variable of vehicles images are loaded\*/*

**setInRotation()**

/\*Acc to the current and next lane and flag nos. it checks at the entry of the junction area whether the car is going to rotate or head straight and accordingly set its centre of rotation('xcentre', 'ycentre') and 'rotationtype'. Also loads the images in 'currentimg' corresponding to lane and flag nos\*/

**PaintCar()**

/\*Paints the car on the frame. Uses AffineTransform utilities to rotate the car\*/

## 5.2 Lane

/\*Adds lanes to the frame\*/

**Lane()**

/\*Takes seven parameters: (x1,y1), (x2,y2) – initial and final coordinates of center of the lane; Laneno – each lane has a distinct number; Roadno – number of the road to which the lane belongs; level – 0 denotes ground level and 1 denotes bridge\*/

**PaintLane()**

/\*Paints Lanes according to level\*/

## 5.3 Road

/\*Creates roads each constituting of two lanes\*/

**Road()**

/\*Creates two Lane type objects and sets the flag value: 0 for horizontal road and 1 for vertical\*/

/\*The lane numbers for the two lanes are consecutive\*/

**PaintRoad()**

/\*Calls PaintLane() of the lanes created\*/

## 5.4 DetectJunction

/\*Detects existence of a junction between any two roads, determines the type of junction(5 types) and accordingly places traffic lights\*/

**DetectJunction()**

/\*Takes two road type objects as arguments, detects junction and its type('type') and sets the center of the junction(x,y)\*/

/\*Creates TrafficLightJunction type object for placing traffic light acc to 'type' value\*/

**PaintJunction()**

/\*Calls PaintTrafficLightJunction for painting traffic lights\*/

## 5.5 TrafficLight

/\*Creates Traffic Light Boxes and displays signal acc to its 'state'(0 – red, 1 – yellow, 2 – green) \*/

**TrafficLight()**

/\*Takes three parameters: x – x coordinate of top left corner of the traffic light box; y – y coordinate of the same; type – type of the box, e.g.1 means box on the top left of the junction, 2 for top right, 3 for bottom right and 4 for bottom left\*/

**PaintTrafficLight()**

/\*Paints the traffic light boxes of the desired type and signals i.e.red, yellow, green according to the value of the variable 'state' using setColor() and fillRect()\*/

## 5.6 TrafficLightJunction

*/\*Encapsulates all the Traffic Lights around a junction depending on the type of junction\*/*

### **TrafficLightJunction()**

*/\*Takes three parameters:x – x coordinate of the center of the junction;y – y coordinate of the same;type – type of the junction. ChangeLight type object 'c' is created which controls the synchronization of the traffic lights at a junction\*/*

*/\*According to the type of junction(+ shaped: four traffic lights ;T shaped: three traffic lights) it creates TrafficLight type objects and places them at required positions\*/*

### **PaintTrafficLightJunction()**

*/\*Based on the 'type' of junction and the value of 'signal' of object 'c' the 'state' of the traffic lights are synchronized and painted on the frame\*/*

## 5.7 button

*/\*Creates artificial buttons for adding vehicles\*/*

### **button()**

*/\*The constructor takes three arguments: xcentre – x coordinate of the centre of the button; ycentre – y coordinate of the same; r – Road type object, each button is associated with a road\*/*

### **findlane()**

*/\*It specifies to which lane of the road is the button associated with\*/*

*/\*Returns 1 for the first lane of the road and 2 for second lane of the same\*/*

### **paintbutton()**

*/\*Paints a (+) at the side of the road\*/*

## 5.8 ChangeLight

*/\*Controls the synchronization of a Traffic Light Box\*/*

*/\*A Thread is associated with its object and at each thread run the value of signal is updated\*/*

Detailed explanation given in Section 4.6

## 5.9 WeightedGraph

*/\*Contains information about the connectivity between vertices(junctions) and the weights of the edges(path)\*/*

### **WeightedGraph()**

*/\*The constructor takes the total number of junctions 'n' as its parameter. Creates an nxn size array 'edges' initialized with zeroes and labels the junctions. The value of the 'edges[i][j]' denotes the weight of edge between i'th and j'th vertex. If its value is zero it means there does not exist an edgebetween the two \*/*

### **size()**

*/\*Returns the total number of vertices(junctions)\*/*

### **setLabel()**

*/\*Associates a label with each vertex\*/*

### **getLabel()**

*/\*Returns the label of the vertex number passed as argument\*/*

### **addEdge()**

```
/*Adds an edge between a pair of vertices('source' and 'target') and associates a weight 'w' with that edge*/
```

```
isEdge()
```

```
/*Returns whether there exists an edge between a pair of vertices*/
```

```
removeEdge()
```

```
/*Removes the edge between a pair of vertices and sets it weight to zero*/
```

```
getWeight()
```

```
/*Returns the weight between a pair of vertices*/
```

```
neighbours()
```

```
/*Returns an array containing those vertices that are next to the given vertex(passed as parameter)*/
```

```
print()
```

```
/*Prints the pair of vertices that has an edge and the associated weight*/
```

## 5.10 Dijkstra

```
/*Returns the shortest path between two junction depending on the weights of the edge(here weight is the distance between a pair of junctions for which an edge exists and the traffic density on that lane)*/
```

```
dijkstra()
```

```
/*Takes two parameters: G – WeightedGraph type object; s – source vertex; and returns the array of shortest path*/
```

## 5.11 Test

Extends JPanel Implements Runnable

```
/*The Test class is the workhouse of the program. It encapsulates the main function and runs a thread*/
```

```
Test()
```

```
/*Coordinates and type of the roads are set. Depending on the position of roads junctions and buttons are detected and traffic lights are placed*/
```

```
/*WeightedGraph type object 'Graph' is initialized*/
```

```
run()
```

```
/*At each thread run:
```

```
    it updates the Graph depending on the traffic density of the lanes and simultaneously changes the path of the car;
```

```
    changes the position of the car according to its updated path;
```

```
    at every junction checks the state of traffic lights thus controlling the flow of traffic;
```

```
    checks for collision between pair of cars and avoids them;
```

```
    generates two unequal random nos between 0 and 99 and if they happen to match the button nos(each button has a unique number) then a vehicle is added;
```

```
    repaints the frame;*/
```

```
addMouseListener()
```

```
/*Whenever the mouse is clicked an event is generated and its coordinates are against the boundary area of each button */
```

```
/*If matched it waits for another mouse click over another button and accordingly a vehicle is added by the user. The first button clicked is the starting point and the second one is the destination*/
```

```
updatepath()
```

```
/*Takes two parameters: i – car number; j – junction at which the car is present. Depending on the current weights of the edges it finds the optimal path for the vehicle from junction j*/
```

```
findclosestjunction()
```

*/\*Each button has a road associated with it. This function takes a button type object and finds the junction closest to the button on than road\*/*

#### **finalpath()**

*/\*The dijkstra() in Dijkstra class returns an array containing junction numbers which is passed as argument to this function. It manipulates the array and generates an array of lane numbers which sets the optimal path for the vehicle\*/*

#### **paintComponent()**

*/\*Paints the car, roads and other components on the frame. Also it displays the numbers of vehicles on a lane between pair of junctions and updates it on each thread run\*/*

## 6 FURTHER IMPROVEMENTS

- Terrain Detection: Through Image Processing Technology one can easily detect roads, bridges, railroads and thus there would be no need of making Road type objects by the user.
- Optimization of Traffic Lights: It can be on the basis traffic density, length of roads and other relevant factors.
- Multiple Lanes: In our program each road has two lanes allowing movement in complementary directions. But it can be extended to include multiple lanes.
- Zebra Crossing: One can also implement Zebra Crossing which is quite easy.
- Dynamic Addition of Roads, Railroads, Bridges in the Terrain.

## 7 REFERENCES

- Wikipedia: Texts on Traffic Simulation and its uses, Graph Theory, and Dijkstra's Algorithm has been taken from Wikipedia.
- Google
- The Complete Reference to Java by Herbert Schildt.
- O'Reilly – Java 2D Graphics by Jonathan Knudsen.

## 8 ACKNOWLEDGEMENTS

- Sun Microsystems for java programming language.
- Coordinators, Programming Club.
- SNT Council, IITK, and other institute authorities concerned.

## 9 RELEVANT LINKS

- About Simulation: <http://en.wikipedia.org/wiki/Simulation>
- JFC Swing [http://en.wikipedia.org/wiki/Swing\\_%28Java%29](http://en.wikipedia.org/wiki/Swing_%28Java%29)
- Graph Theory [http://en.wikipedia.org/wiki/Graph\\_theory](http://en.wikipedia.org/wiki/Graph_theory)
- Dijkstra's Algorithm  
[http://en.wikipedia.org/wiki/Dijkstra%27s\\_algorithm](http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)  
<http://renaud.waldura.com/doc/java/dijkstra/>
- Example of Traffic Simulation:  
<http://www.phy.ntnu.edu.tw/oldjava/Others/trafficSimulation/applet.html>