Traffic Simulation Project

PTS report submitted to the
IBO Department
Leonardo da Vinci Engineering School

Junhui LI

Anas Hadhri

Johann de Soyres

Summary

1. Introduction

Almost all the populated countries especially under developed countries are facing traffic problem in their major big cities like Paris in France. Traffic is increasing at a rapid pace on the roads proportional to the increase in population. The traffic problem is faced by all big cities in all countries. As the traffic increases it causes traffic congestion. Cities are facing number of serious problems like wastage of time, depression, pollution, risk of accidents, and other serious problems. These problems cause an extra burden to the society and business. According to news, in 2002, 750 million Euros loss has been reported in Greater Copenhagen due to traffic problem.

People are using cars or other vehicles that basically use roads for moving to their destination. This trend is increasing day by day. Current analysis of big cities shows that the number of cars on the roads is increasing tremendously and the available resources are limited. Proper handling of traffic seems to be the market demand. Intelligent optimization algorithms are the necessity of an optimal control of traffic. Traffic problem is a well-known optimization problem from a decade or so. They may be helpful in enhancing the capacity of the roads and traffic flow too. Several intelligent algorithms are proposed for achieving this purpose as Dijkstra algorithm or A star algorithm. This led us to the problematic around traffic simulation.

Before talking about our project, we will begin with a short state of art and the architecture on which our solution was built. After that, we will talk about our workflow. Then there will be an entire description of our solution. Finally, we will conclude this report with a feedback.

2. Problematic:

How can we simulate traffic flows in big cities?

3. State of art

3.1 Traffic simulation models

Traffic simulation is the mathematical modeling of transportation systems thanks to computer software which can help plan, design, and operate transportation systems. Simulation of transportation systems started over forty years ago and it 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 help in their management of transportation networks.

There are four kinds of traffic simulation model:

- **Microscopic**
- **Macroscopic**
- **Mesoscopic**
- **Metascopic**

Microscopic models are designed to simulate single vehicle's behaviors. A microscopic model of traffic flow attempts to analyze the flow of traffic by modeling driver-driver and driver-road interactions within a traffic stream. This traffic stream respectively analyzes the interaction between a driver and another driver on road and of a single driver on the different features of a road.

Figure 1 : Microscopic simulation view

The dynamic variables of the models represent microscopic properties like the position and velocity of single vehicles. Many studies and researches were carried out on driver's behavior in different situations like a case when he meets a static obstacle or when he meets a dynamic obstacle.

Macroscopic models describe the dynamic of traffic flows.

These models are applicable in the development of dynamic traffic management and control systems.
They are principally designed to optimize the traffic system and can be used to estimate and predict average traffic flow operations.

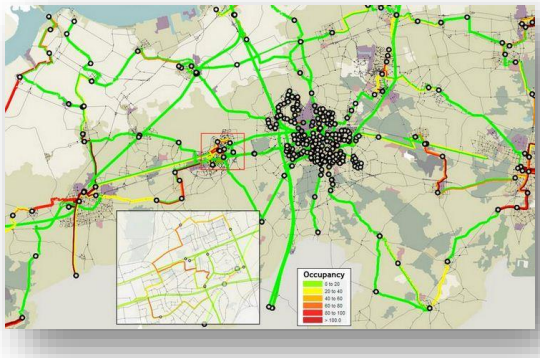


Figure 2 : Macroscopic simulation view

The main advantage of Macroscopic traffic models over microscopic models is the significantly lower computational costs due to lower complexity.

Mesoscopic models study vehicles which are similar in small groups, for example several cars. Mesoscopic traffic flow models were developed to fill the gap between the family of microscopic models that describe the behavior of individual vehicles and the family of macroscopic models that describe traffic as a continuum flow.

Figure 3 : Mesoscopic simulation view

Traditional mesoscopic models describe vehicle flow in aggregate terms such as in probability distributions. However, behavioral rules are defined for individual vehicles.

Metascopic models are a new kind of traffic models. With these models traffic flow can be modelled as liquids in a bathtub with taps and drains. Thanks to this level of abstraction, the description of network traffic operations turns out simple.

3.2 Applications

Traffic simulation can be very useful in many areas such as traffic prediction, assessing performance of new transportation infrastructures and analyze consequences of important traffic flows (air quality, accident rates, etc). All type of vehicles can be considered to simulate traffic: cars, trucks, planes, boats, cyclists, etc.

Here is a list of the most well-known traffic simulation packages that are used in various traffic simulation software around the world:

- **SUMO**

  SUMO (Simulation of Urban Mobility) is an open source, highly portable, microscopic road traffic simulation package designed to handle large road networks.

- **Quadstone Paramics Modeller**

  Quadstone Paramics is a modular suite of microscopic simulation tools providing a powerful, integrated platform for modelling a complete range of real world traffic and transportation problems.

- **Treiber's Microsimulation of Road Traffic**

  Treiber's Microsimulation is a software project used in research of traffic dynamics and traffic modelling.

- **Aimsun**

  Aimsun is a simulation package that integrates three types of transport models: static traffic assignment tools; a mesoscopic simulator; and a microsimulator.

- **Trafficware SimTraffic**

  SimTraffic is a simulation application part of Trafficware's Synchro Studio package. It serves as a traffic simulator for Trafficware's Studio, which also includes traffic lights synchronization application.

- **CORSIM TRAFVU**

  CORSIM TRAFVU provides animation and static graphics of traffic networks, using the CORSIM input and output files created by a licensed user of TSIS CORSIM.

4. Workflow

- **IntelliJ IDEA**

  IntelliJ IDEA is a commercially available Java Integrated Development Environment (IDE) tool developed by JetBrains Software. It has the following advantages:

  - ✓ Debugging
  - ✓ Autocomplete
  - ✓ Refactoring
  - ✓ Detecting duplicates
  - ✓ Support development tools

- **GitHub**

  GitHub is a web-based hosting service for version control using Git. It is mostly used for computer code. It offers all of the distributed version control and source code management (SCM) functionality of Git as well as adding its own features. We have multiple developers working in parallel. So, a version control system like Git is needed to ensure there are no code conflicts between us. Version control system allows us to revert and go back to an older version of the code.

Sometimes several projects which are being run in parallel involve the same codebase. In such a case, the concept of branching in Git is very important. The above is why we want to use GitHub.

- **Skype**

  Skype is a VoIP service, which uses the Internet to allow people to make and receive free voice and video calls online. We used skype for daily meeting discussions.

- **Agile software development**

  Agile software development is an approach to software development under which requirements and solutions evolve through the collaborative effort of team members. It means adaptive planning, evolutionary development, empirical knowledge, and continual improvement, and it encourages rapid and flexible response to change.

  At the regular meeting, we report progress, exchange our points of views and discuss about upcoming tasks. We use GitHub to record and share our code. On GitHub we built features branches to save code at different schedules in order to work at the same time on different functionalities. Then we integrate these new features into the master branch thanks to git merge.

5. Traffic Simulation Application

To simulate traffic congestion in big cities we choose to implement our own macroscopic traffic model. Macroscopic models are the more relevant to simulate traffic flows in an entire big city. As we say before they need less computational resources than other models when many vehicles are taking into account. We choose to build our traffic model from scratch because it allows us to create the best model for our needs and to fully understand the way it works.

The purpose of our model is to simulate traffic flows in the next decades in big cities. For this we decide to build a Java application with MVC design pattern.

Model-View-Controller is an architectural pattern commonly used for developing user interfaces that divides an application into three interconnected parts.

- Model - Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.
- View - View represents the visualization of the data that model contains.
- Controller - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

This is done to separate internal representations of information for the user. The MVC design pattern allows efficient code reuse and parallel development. MVC pattern has many advantages:

- Multiple developers can work simultaneously on the model, controller and views.
- MVC enables logical grouping of related actions on a controller together. The views for a specific model are also grouped together.
- The very nature of the MVC framework is such that there is low coupling among models, views or controllers.
- Because of the separation of responsibilities, future development or modification is easier and scalability of the product is increased.
- Multiple views for a model

We firstly focus on how the city could be modelled.

5.1 Data Architecture

Modeling a city has been quit a challenge for us. Cities can be very differently structured and very complex.

Figure 4 : Paris

Figure 5 : Los Angeles

However, the trend is starting to disappear with modern cities like Los Angeles, New York or Dubai where roads and buildings form rectangular structures.

Therefore, we choose to consider a city as a **matrix of nodes**. A **node** can be a building, a piece of road, a piece of Park, etc. Each node has characteristics such as if it is occupied, coordinates (x,y), neighbors, if it contains an accident, a taxi, a bus stop, etc. For our project we decided to take **Manathan** as city for the whole project. Manathan is a good example of how cities will be structured in the future. It has a very simple structure. To import the city into our application we built a txt file where all characteristics are encoded in numbers and characters. As you can see below each node is encoded by a character or by a number. On figure 6 you can see the map we used to know where to place roads, buildings and central park. Then we put on our own some bus stops, taxi stops, accidents, traffic lights and stops.

Figure 6 : Manathan map

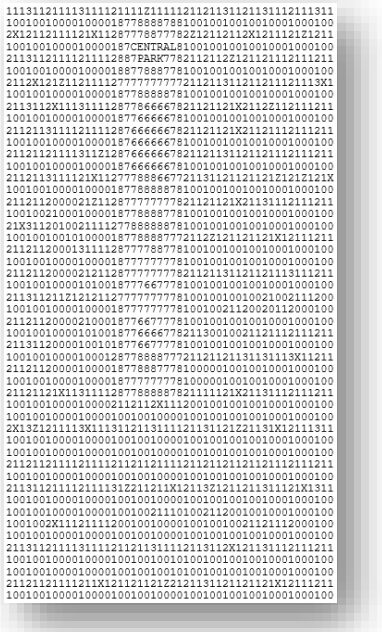


Figure 7 : Manathan encoded map

Now we can build a matrix of nodes thanks to this txt file. We read the file, build for each character or number a node with characteristics corresponding to it.

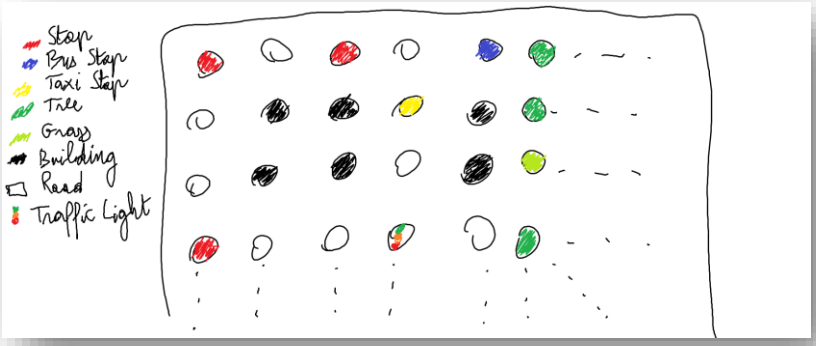


Figure 8 : Matrix of nodes

Now our application contains a matrix of nodes which represent a part of Manathan. So how can we simulate traffic flows in this city? A naïve approach would be to put many cars in the city and give them random itineraries. This approach raises many issues. First, this solution would compute many resources because for each car it would be necessary to choose the next position. But another and more important issue is the difference between this approach and the reality. When you drive a car

you never choose your path randomly. If you do that you don't take into account factors like traffic light, stops or distances in your path's choice. In result traffic flows would be very weird and the major axes in our city will not be used as they should be in the reality.

So, we know that that this approach is not the right solution. To resolve this problem, we had to ask ourselves how a path's choice could be the more realistic possible? How a path's choice would be like for a driver in the future? Well the answer is based on **GPS apps**. GPS has in the last years been intensively used as receivers for directions and as navigations app that tells us where we are and how to move between points. They have also been used to make digital maps. However, GPS apps are not only used by civilians but also by different organizations for different purposes. For example, it is useful to pilots, surveyors, farmers, boat captains, military and scientists among others. Over the years GPS apps have become more accurate and efficient. Among the most famous GPS apps we found **Google maps**, **Waze**, **Sygic** and **Here WeGo**.

Today many drivers use GPS apps to guide them to their destination. And we can highly suppose that in the future everyone will uses GPS apps when they drive. Gps apps used pathfinding algorithms to choose a path for the driver. So, if we want to simulate traffic flows we need to place cars in the city, choose a destination and then apply a pathfinding algorithm for every car.

5.2 Pathfinding algorithms

Which pathfinding algorithm should we use?

There are many popular pathfinding algorithms. To find the shorthest path, Astar and Dijkstra are the more adapted. Pathfinding algorithm are generally applied on graphs. A **graph** is a data structure which consists of a finite set of vertices.

The graph contains:

- vertices, or nodes, denoted in the algorithm by v or u;
- weighted edges that connect two nodes: (u, v) denotes an edge, and w(u, v) denotes its weight. In the diagram on the right, the weight for each edge is written in gray.

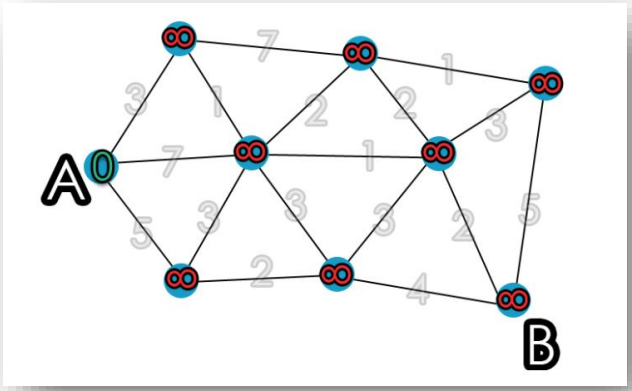


Figure 9 : graph

Dijkstra's algorithm

Dijkstra's algorithm finds a shortest path from a single source node, by building a set of nodes that have minimum distance from the source in a graph. This is done by initializing three values:

- dist, an array of distances from the source node s to each node in the graph, initialized the following way: dist(s) = 0; and for all other nodes v, dist(v) = ∞. This is done at the beginning because as the algorithm proceeds, the dist from the source to each node v in the graph will be recalculated and finalized when the shortest distance to v is found
- Q, a queue of all nodes in the graph. At the end of the algorithm's progress, Q will be empty.
- S, an empty set, to indicate which nodes the algorithm has visited. At the end of the algorithm's run, S will contain all the nodes of the graph.

Then the algorithm proceeds as follows:

1. *While Q is not empty, pop the node v, that is not already in S, from Q with the smallest dist(v). In the first run, source node s will be chosen because dist(s) was initialized to 0. In the next run, the next node with the smallest dist value is chosen.*

2. *Add node v to S, to indicate that v has been visited.*

3. *For each new adjacent node u, if dist(v) + weight (u, v) < dist(u), there is a new minimal distance found for u, so update dist(u) to the new minimal distance value.*
   Otherwise, no updates are made to dist(u).

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

Figure 10 : Dijkistra pseudo code

At the end you get all the possible and shortest paths from the source node in the graph. To choose the path you are interested in, you just have to specify the destination.

Astar algorithm

A* is a computer algorithm that is widely used in pathfinding and graph traversal. This pathfinding algorithm is very different from Dijkistra. With Astar you can compute only the path you are interested in.

Astar works by maintaining two lists; the open list and the closed list. The open list's purpose is to hold potential best path nodes that have not yet been considered, starting with the start node. If the open list becomes empty, then there is no possible path. The closed list starts out empty and contains all nodes that have already been considered/visited.

The core loop of the algorithm selects the node from the open list with the lowest estimated cost F to reach the goal. **A cost is a value that include distance and time to reach a next node**. Here F = G + H.

- F is the total cost of the node.
- G is the distance between the current node and the start node.
- H is the heuristic—estimated distance from the current node to the end node. In our case we use an Euclidean heuristic.

$$d(X,Y) = \sqrt{\sum_{i=1}^{n}(y_i - x_i)^2}$$

Figure 10 : Euclidean heuristic

This is done by initializing three values:

- Q, which is the open list, a queue of all nodes in the graph.
- S, which is the closed list, an empty set.
- The F cost for the source node to zero

Then the algorithm proceeds as follows:

1. *While Q is not empty, Find a node q in Q with the lowest F cost and pop it from Q.*

2. *For each neighbor n of q, if n is the destination stop the algorithm else set H cost, G cost and F cost for the node n.*

   If a node with the same coordinates than n is present in Q and has a lower F cost then skip n.

   If a node with the same coordinates than n is present in S and has a lower F cost then skip n.

   Else add n to Q.

3. *Push q in S.*

As we can see, Astar algorithm give us directly the path we are interested in.

```
initialize the open list
initialize the closed list
put the starting node on the open list (you can leave its f at zero)

while the open list is not empty
    find the node with the least f on the open list, call it "q"
    pop q off the open list
    generate q's 8 successors and set their parents to q
    for each successor
        if successor is the goal, stop the search
        successor.g = q.g + distance between successor and q
        successor.h = distance from goal to successor
        successor.f = successor.g + successor.h

        if a node with the same position as successor is in the OPEN list \
            which has a lower f than successor, skip this successor
        if a node with the same position as successor is in the CLOSED list \
            which has a lower f than successor, skip this successor
        otherwise, add the node to the open list
    end
    push q on the closed list
end
```

Figure 11 : Astar pseudo code

Graph structure

Before talking about our choice between these two algorithms we need to discuss about a more important problem. As we say before, pathfinding algorithms are generally applied to graphs and not matrix. We built a graph from our city.

The challenge was to extract roads from the matrix. We don't want useless nodes like buildings in our graph because they will never be relevant for the path search. So, what we tried to do was to build a graph which provides an efficient time and space complexity without buildings in it.

First, we build a list of all nodes which are not occupied. Then for each node in the list we create two arrays:

- The first one contains **all the index (index the initial list) of neighbors**
- The second one contains **all the costs for the corresponding neighbors**

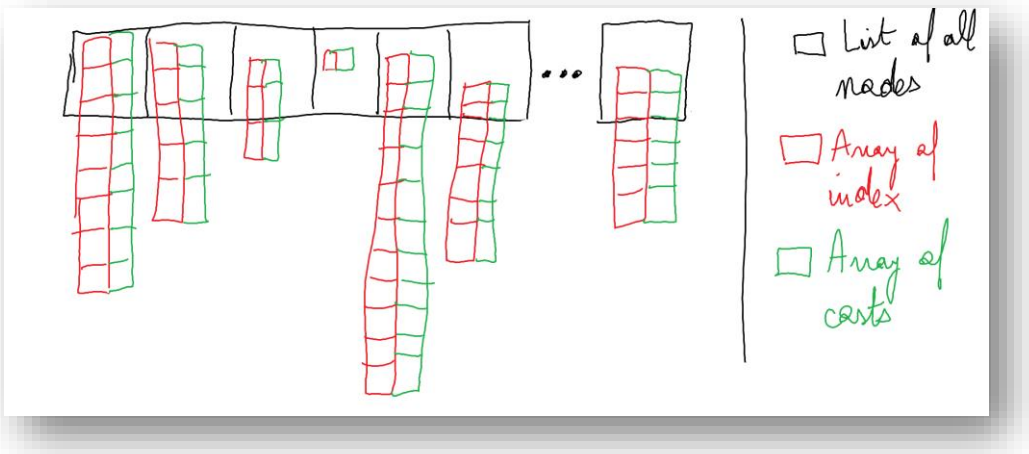


Figure 12 : Graph scheme

With this data structure, we can easily get neighbors and their costs for a given node. We can also easily browse our graph. For example, for the path [node 1 → node 3 → node 5 → node 6] we have:

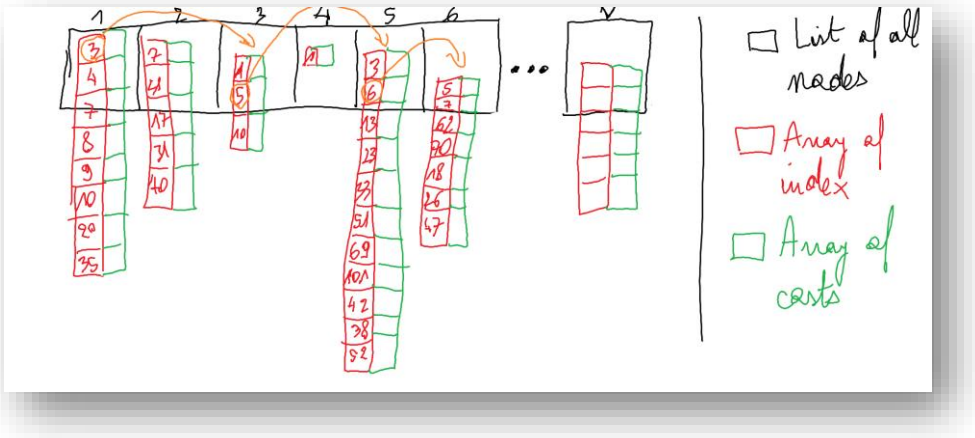


Figure 13 : Graph traversal

Since we build the graph from the matrix, an array of neighbors can contain minimum 1 index and maximum 8 index, same for an array of cost. At this point of the project we did a mistake. Instead of putting every node which is not occupied in our graph we could have just put intersections. With this second approach we would have been able to handle huge matrix and to reduce time and space complexity at the same time. But then it would have been more complex to implement it. Unfortunately, we realized this too late and we decided to continue with our initial structure in order to save time for the project.

Dijkstra's algorithm vs Astar algorithm

E: number of edges

V: number of vertices

Parameters	A* algorithm	Dijkistra's algorithm
Time complexity	O(E)	Original Dijkistra – O(n^2) Binary min-heap Dijkistra -O(E*log(V)) Fibonacci heap Dijkistra – O(E + V*log(V))

Heuristic	F = G + H	F = G
Dynamic Version	Exists	Doesn't exists

This comparison shows us an important point. In fact, Dijkstra's algorithm can't be implemented in a dynamic way. This means that Dijkstra's algorithm is not capable to take into account dynamic factors. For instance, if we launch the algorithm and then an accident happen in the graph, Dijkistra will not take into account the accident and calculate one more time all the paths. But Astar is capable of that. If an accident happen in the city Astar will noticed it with the H cost.

Now if we look at times complexities, we can replace E (number of edges) by V*8 which is equivalent to V.

Parameters	**A* algorithm**	**Dijkistra's algorithm**
Time complexity	O(V*log(V))	Original Dijkistra – O(V^2) Binary min-heap Dijkistra - O(V*log(V)) Fibonacci heap Dijkistra – O(V + V*log(V))

Astar complexity is equivalent or better than Dijkistra complexity. Because of all these facts, we choose Astar as pathfinding algorithm for our project.

5.2 Views

A view for an application is an interface. The user can interact with the application thanks to this interface. To build our view, we use the library Java SWING.

Java Swing

Java Swing is a GUI widget toolkit for Java. It is part of Oracle's Java Foundation Classes, an API for providing a graphical user interface (GUI) for Java programs.

Swing was developed to provide a more sophisticated set of GUI components than the earlier Abstract Window Toolkit. It has more powerful and flexible components than AWT. In addition to familiar components such as buttons, check boxes and labels, Swing provides several advanced components such as tabbed panel, scroll panes, trees, tables, and lists. Unlike AWT components, Swing components are not implemented by platform-specific code. Instead, they are written entirely in Java and therefore are platform-independent.

View classes

We create several view classes: **Display, GameView, Button, Grille** and **Cityframe**.

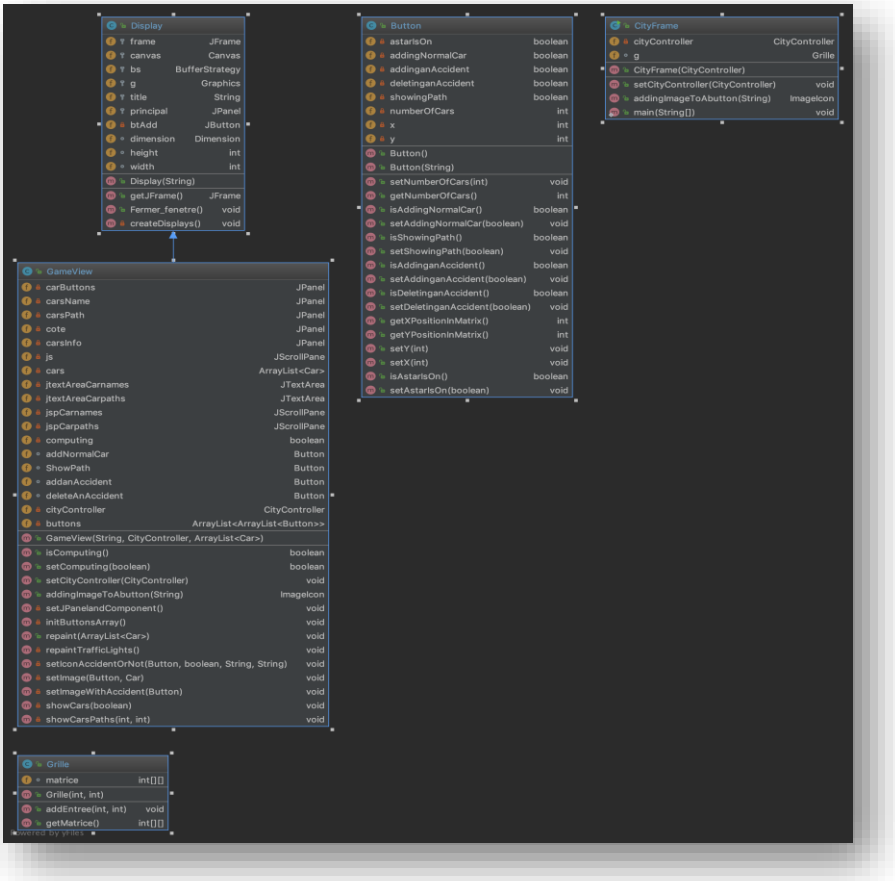


Figure 14 : View classes

The goal of GameView is to display a window with two panels:

- A panel on the left side for our matrix
- A panel on the right side to interact with the user (information and functionalities)

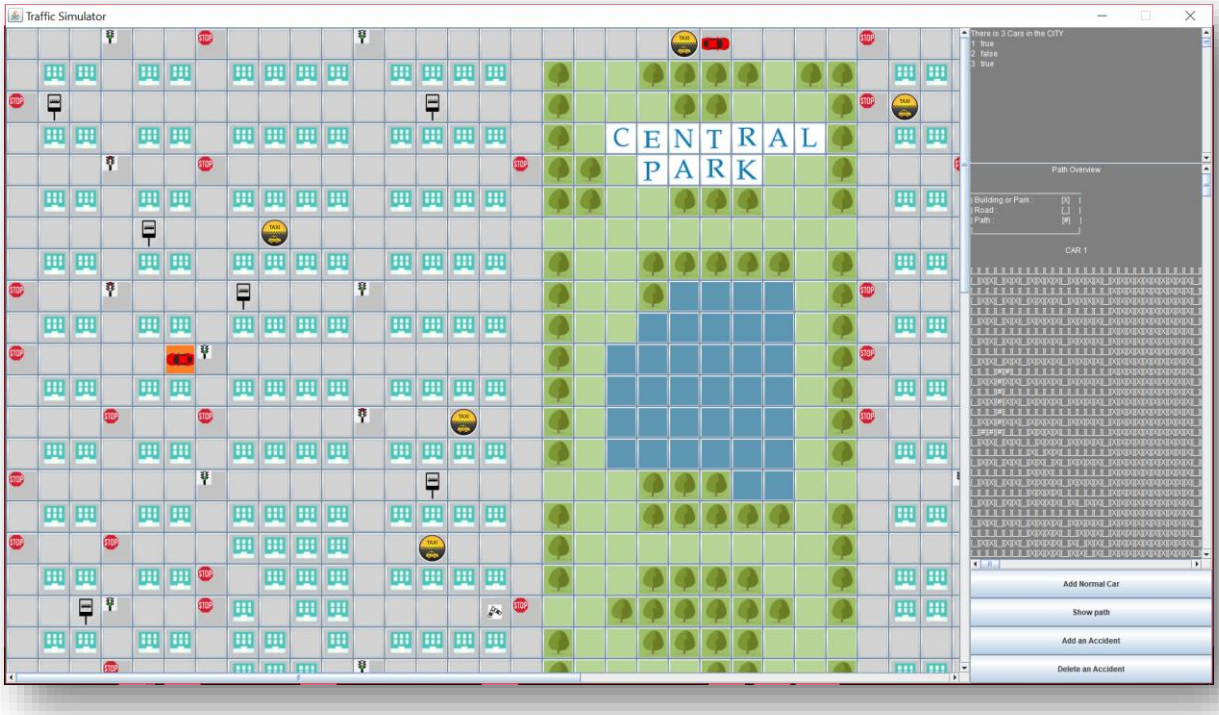


Figure 15 : Application interface

To display the matrix in the left-side panel, we built a matrix of buttons thanks to the initial matrix of nodes. This matrix of buttons was created thanks to the **classes Grille**. We also design our own **Button classes**. Our button could contain a new car, an accident, a Bus stop, etc.

After that we needed a component to manage the interface, the pathfinding algorithm, cars, the graph and the city together.

5.3 Controller

What is a Controller? Controller is a software program that manages and directs the flow of data between entities. In our game we have 4 controllers: **Graph, Astar, CityController ,GameManager.**

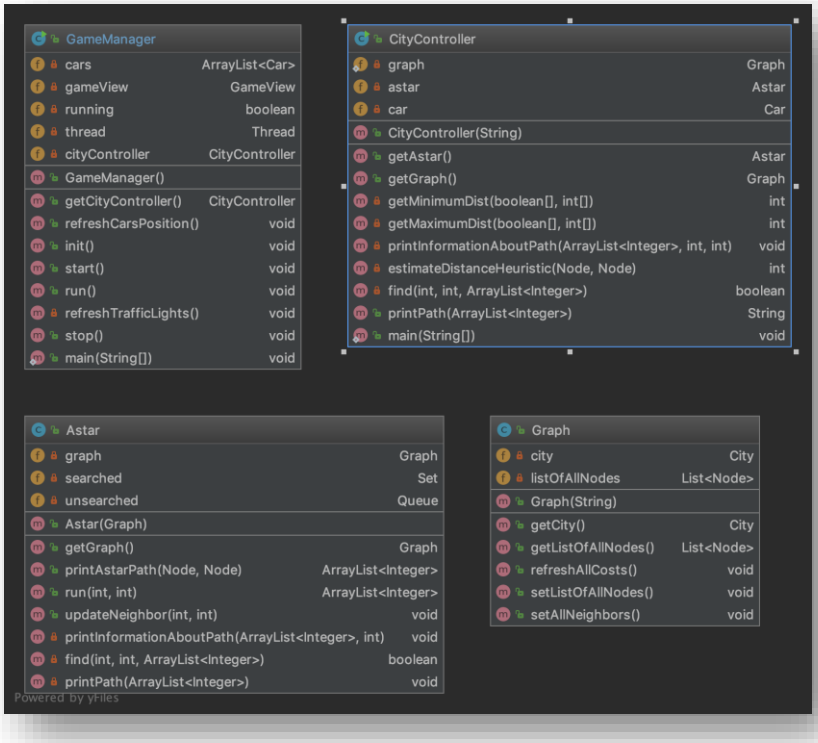


Figure 16 : Controller classes

Now let's speak in detail about our main controller which is GameManager that is responsible for **starting the application**.

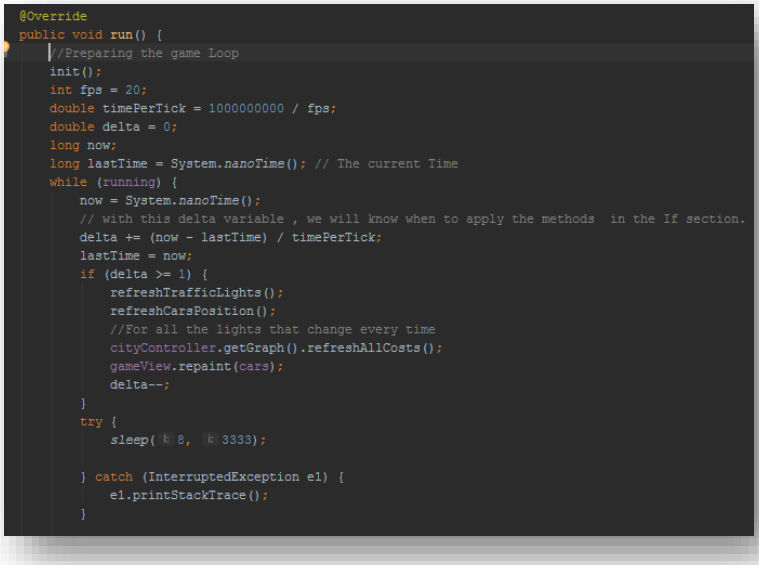
GameManager Controller

GameManager is a thread program. Let's briefly define what is a Thread. A thread is a smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. GameManager is responsible for simulating the traffic of cars, displaying cars and the city. It controls the application from the beginning to the end.

Why did we choose GameManager to be a Thread?

We don't want to interrupt the application and the traffic simulation when the user decides to add a car for example. So, we have our GameManager that handles our traffic simulation and the Main Thread that is responsible for the user interactions.

Let's speak about the **run method** of the Thread GameManager which is responsible for starting the game.



```
@Override
public void run() {
    //Preparing the game Loop
    init();
    int fps = 20;
    double timePerTick = 1000000000 / fps;
    double delta = 0;
    long now;
    long lastTime = System.nanoTime(); // The current Time
    while (running) {
        now = System.nanoTime();
        // with this delta variable , we will know when to apply the methods  in the If section.
        delta += (now - lastTime) / timePerTick;
        lastTime = now;
        if (delta >= 1) {
            refreshTrafficLights();
            refreshCarsPosition();
            //For all the lights that change every time
            cityController.getGraph().refreshAllCosts();
            gameView.repaint(cars);
            delta--;
        }
        try {
            sleep( 8,  3333);

        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }
```

Figure 17 : run method

This method contains the game loop. The game loop is capable to handle time. So, we are not facing the problem that in slower hardware the game runs slower and on faster hardware faster. In our GameManager, we are running on a steady **20 fps.** Fps is an abbreviation for Frames Per Second. In the context of the above implementation, it is the number of times the methods in the if section are called per second.

In the game loop, we initialize the controller of the city (City controller Class) and we display the GameView interface (**init method**).

Then we refresh the state of each traffic light. A traffic light can be Red or Green. Every traffic Light has his own timer. After reaching the timer Max, the state of the traffic light changes (**refreshTrafficLights method).**

After that, we refresh the cost to go from one position to the next position. We are doing this while calling a method in the graph Controller (**refreshAllCosts).** This cost could change when accidents happen for example.

In the game loop we are refreshing all the cars Position. In fact, the position of existing cars will change in the case that the **timer** of the car is equal to the **timer max**. **refreshCarsPositions is** the method responsible for this and it is communicating with the model class named **Car**.

Our Game Manager Controller is communicating with the view class named GameView. And it is responsible for showing all the images into their position in the user interface. (**Repaint method)**

User Interaction

In our software, the user has a lot of features to interact with the traffic.

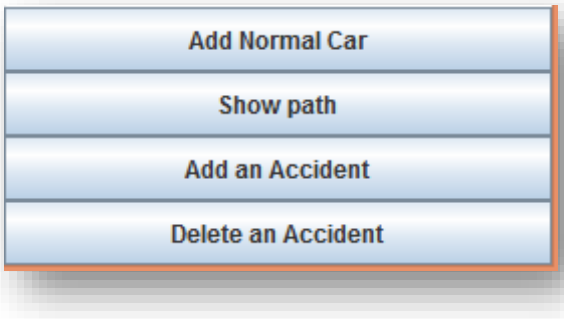


Figure 18 : Functionalities

Action Listeners

ActionListeners are used for handling action events. For all the interaction of the user, we developed Action listeners.

1- Add Normal Car button

After clicking on this button, the user has to click on a position in the city, so a new Car will be added in this position. In the action listener of this feature, we are initializing a new object Car, we are refreshing the Array list cars that contains all the car objects in the city and we increment the number of cars that occupy the button.

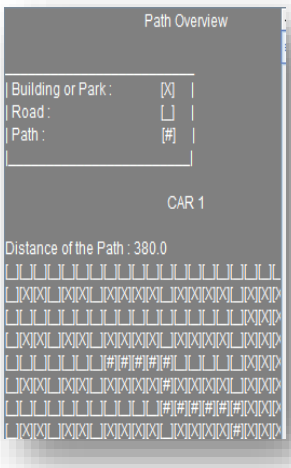
2- Add an accident

After clicking on this button, the user has to click on a position in the city and a new accident will be added in the select position. In the action listener of this feature we specify for the current Node that corresponds to the Button the information of adding an accident. So, we call the **method setAccident** with a **true** Boolean.

3- Delete an accident

As adding an accident, after clicking on this button the user can select an accident to delete. In the action listener of this feature we specify for the current Node that corresponds to the Button the information of deleting an accident. So, we call the **method setAccident** with a **false** Boolean.

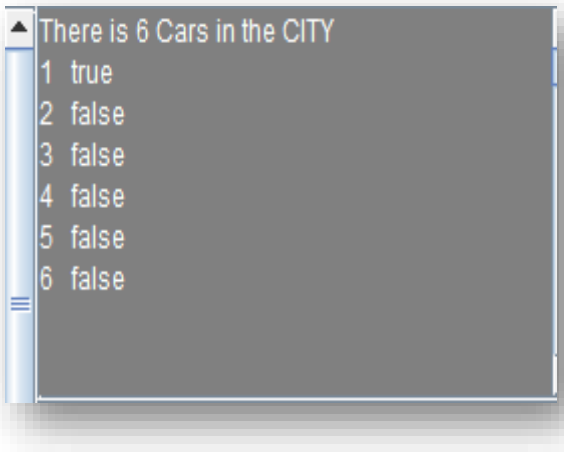
4- Show Path



With this button, the user can follow the path for the selecting car. The user has to select at first the car, then click on this button. The path is dynamic, and the user can also visualize the Identifier of the car and the distance path. When the user clicks on this button, we are calling a method in gameView class named showCarsPath. This method uses information in the class Node and Car in order to know how to display the path.

Figure 19 : show path panel

5- Information about cars



The user can visualize the cars which are in the city. As we can see on the image above, we have the identifier of the car and boolean that takes true or false. This boolean variable change to true when the user selects a car in the city.

Figure 20 : Cars information panel

This is handling by the method **showCars** that it is called in the method repaint().

6. Feedback

Thanks to our collaboration in the project we created a Traffic simulator application which is able to simulate traffic flows in big cities. We would like to thank our school mentor **David Dupuis** who gave us a lot of good advices.

This project gave us the opportunity to learn a lot of concepts and apply them:

- Advanced data structures like graphs, queues and stacks
- The Model – View – Controller design pattern
- Pathfinding algorithms
- Agile methodology
- Resolving conflicts between team members
- Java 8 / Java Swing

There is a lot of features that can ameliorate our project:

- Undirected graph to directed graph. It means that it will be possible to have directed and undirected roads at the same time.
- Dynamic version of Astar
- 3D user interface
- Add some scenario cases (calling an ambulance for an accident)
- Add velocity for each car

Our application could be used by towns halls which want to analyze consequences of new infrastructures on traffic flows, understand the origin of regular traffic congestion and predict traffic flows.