UNIVERSITAT JAUME I

Development of a distributed system for real time traffic analysis using agents

by

Pablo Jiménez Mateo

A thesis submitted in partial fulfillment for the degree of Master in Intelligent Systems

in the
Escola Superior de Tecnologia i Ciències Experimentals
Department of computer engineering

November 22, 2016

Declaration of Authorship

I, Pablo Jiménez Mateo, declare that this thesis titled, ‘Development of a distributed system for real time traffic analysis using agents’ and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ________________________________

Date: ________________________________

"Given the pace of technology, I propose we leave math to the machines and go play outside."

Calvin, Bill Watterson's creation.

Abstract

Nowadays, technology provides researchers a lot of data to work with. This data usually gives a good overview of how the people behave and is used for marketing purposes, but it can also help improving people lives.

Smart cities and smart transport take advantage of this data to make the city more efficient, and the mobility more fluid. By changing traffic lights according to the needs, or reroute the traffic through a different path, the overall time spent on the road is reduced, and pollution is minimized.

In this thesis, this problem is addressed, first a complete distributed multi agent system is developed to work with it as a testbed, then existing routing algorithms are discussed and a new smart algorithm is proposed. Finally, it is shown that the proposed algorithm behaves better, making the traffic on the road network more fluid.

Acknowledgements

I want to thank Vicente Ramón Tomás López for its patience and recommendations as my supervisor. I also want to thank my lab partner Víctor Pérez Miralles, we spent so much time together in the lab, discussing ideas and helping each other. I want to thank Krakatoa, the mighty tea maker, I couldn't have done it without it.

And, the most important thank you goes to my family, they have been helping me for a long time before this thesis, with their unconditional support and patience, and I am sure that will keep on with it, I would surely not be were I am without them.

Contents

List of Figures

List of Tables

Abbreviations

API	**A**pplication **P**rogramming **I**nterface
GPS	**G**lobal **P**osition **S**ystem
GUI	**G**raphical **U**ser **I**nterface
IoT	**I**nternet **o**f **T**hings
MAS	**M**ulti **A**gent **S**ystem
WORA	**W**rite **O**nce **R**un **A**nywhere

Dedicated to my family

Chapter 1

Introduction

In this first chapter, the motivation of the project and its main objectives are stated. After that the time planification, employed technologies and the structure of the rest of the thesis are presented.

1.1 Motivation

Nowadays, with the penetration of new technologies such as smartphones and self driving cars, it is possible to share information about the state of the traffic on the transportation networks. This is very useful to have real time path finding algorithms that avoid congestion.

Furthermore, vehicle to vehicle and vehicle to infrastructure communications has been researched for a long time [1] and is still being researched nowadays [2], and this is a direct application of it.

The motivation of this project is to help research areas that study how autonomous and automatic systems can achieve a better network mobility thus reducing pollution. Developing a Multi Agent System to test different routing algorithms and even proposing a brief study on existing algorithms will help researchers to get to that goal. Multiple rerouting algorithms will be tested based on [3] and results will be analyzed based on the real time state of the traffic.

1.2 Objectives

The main goals of the final master project are:

- Development of a simulator to test traffic routing algorithms
- Analyze the existing algorithms for traffic routing
- Propose an algorithm that reacts to the traffic in real time
- Study and analysis of the proposed algorithm

1.3 Planification

Task	**Planned hours**	**Goal**
Study and installation of the JADE library	10 hours	Understand how the library JADE works
Development of a graph to represent the urban network	5 hours	Complete the classes and files needed to represent the network
Development of a communication protocol and its ontologies	20 hours	Understand how the inter agent communication will be done
Development of agents	-	-
- Development of the first mobile agents with JADE	15 hours	Implement the vehicles
- Development of the behavior of the vehicles	15 hours	Implement the logic of the vehicles
- Development of segment agents	15 hours	Implement the segments
- Development of the agent that keeps the time	10 hours	Implement a reliable and deterministic model of time
- Development of the agent that launches the events	10 hours	Implement an agent able to read events from file and launch them to the simulator
Development of the graphical user interface	25 hours	Implement a graphical user interface to view the results in real time
Development of routing algorithms	35 hours	Implement various routing algorithms
Testing and optimizing the application	40 hours	Stress testing the application
Gathering of results	10 hours	Running simulations and creating the graphs
Documentation	10 hours	Documentation of the communication protocols, ontologies and classes
Writing the thesis	80 hours	Writing of the thesis
TOTAL	**300 hours**	

TABLE 1.1: Time planification

1.4 Employed technologies

The language of choice for this project was Java [4], Java is an all purpose programming language that can be run in pretty much any device following the WORA [5], from desktop operative systems to mobile devices.

This allows us to create a distributed client for this application that will be able to run in an Android or iOs device, making it way easier for all the users to benefit from it without huge limitations.

The GUI is made using the Java swing library [6] to keep dependencies at a minimum, it is a really powerful library that has been used to allow the user to control the desktop application.

The maps image has been taken from a screen capture of Google Maps, the general area of the province of Castelló.

All the communication between nodes (cars and road side units in this scope) are made in a distributed way using the standalone library JADE [7]. JADE is a software framework for the development of applications using the agent paradigm and distributed communication. It complies with the FIPA specification [8] and also comes with a few graphical tools to make the debugging and development of a distributed application easier, since it is a really difficult task due to its nature.

The test files and the graphs have been made in Python, using the libraries Matplotlib and Numpy.

1.5 Structure

In the Chapter 2 of this thesis, a brief overview on the state of the art is done. There a little bit of information and context is done about Smart cities and Smart transport, also a description is done about which routing algorithms that were evaluated for this MAS, and the benefits of each one, as well as some information about how they work.

In Chapter 3 the Multi agent system is explained, all its models, classes, agents and ontologies are described there.

Chapter 4 describes the tests that have been performed in the MAS, it also describes the graphics performed, it is shown that the proposed algorithm behaves better than the static ones.

Finally, in Chapter 5 a little bit of personal opinion, faced challenges and future work is discussed.

Chapter 2

State of the art

This chapter gives an overview of previous work, and how this thesis fits within those studies.

2.1 Smart cities

In the last few years, smart cities have been a hot topic in research [9] given the quantity of available data provided by the internet of things [10] and big data [11] and the newest infrastructures that allow cities to track activities in real time. This leads to the use of this data to improve the quality of life of the citizen, employment[12],saving on unused services when they are not necessary and limiting the pollution by making smart transport more fluid.

Some cities are already taking advantage of this technology [13], Amsterdam already monitors traffic in real time and shows information about the best route to take, Barcelona has already optimized bus routes through the center to optimize the number of green lights they encounter.

Smart cities also monitor pedestrian activities to know when to dim lights on the night, or when to put a traffic light green if no one is getting to that intersection. The main concern of a smart city is to save energy, and make the quality of their citizens life as good as possible.

Smart cities is a general idea of which smart transport is only a small part, this thesis focuses on that part, trying to minimize the time spent on the road and thus, the generated pollution.

2.2 Smart transport

Smart transport is a topic that only recently has started to be researched. [14] defines Smart transport as „adequate human–system symbiosis to realize effective, efficient and human-friendly transport of goods and information.“, I would add that smart transport also focuses on minimizing pollution and that is exactly what this thesis is focusing on, a protocol that will make transportation effective, efficient and greener while being human-friendly.

As seen in this review [15] the criteria to find the best protocol is very varied, it can be based on social, economic, environmental or even land usage. Our protocol is focused on minimize the network congestion and thus minimizing the total added time of travel times.

This MAS helps analysts to test different routing algorithms in a distributed way, and mix different algorithms within the same simulation. I believe that this tool will be really useful for anyone that wants an initial test since its usage and output are so easily learned.

2.3 Routing algorithms

In this chapter, well known routing algorithms are proposed and studied.

2.3.1 What is a routing algorithm

A routing algorithm is a deterministic program that finds a path between an origin and a destination. Usually the network where the path has to be extracted from is a graph, directed or undirected.

A basic graph has two main components, vertices and edges. In the context of this thesis, the vertices are traffic intersections and the edges are the roads that connect them.

Edges usually come with a weight, it is a value that represents the cost of traversing that edge, this could be anything from kilometers to time. Graphs can even have more than one kind of weight depending on what we intend to optimize.

In the context of this thesis, the roads have their length and the maximum allowed speed, this allows us to use the most simple metrics on traffic, time and distance.

A routing algorithm takes advantage of all this information and finds the minimum path depending on the metric we want to optimize, in our case time or distance.

In this thesis we are going to focus on the most simple yet effective path finding algorithms.

2.3.1.1 Dijkstra's algorithm

Dijskstra's algorithm is a well known iterative algorithm that finds the shortest path between two nodes in its most simple version. This is the most famous path finding algorithm and has been deeply studied [16] [17].

Its pseudocode is the following:

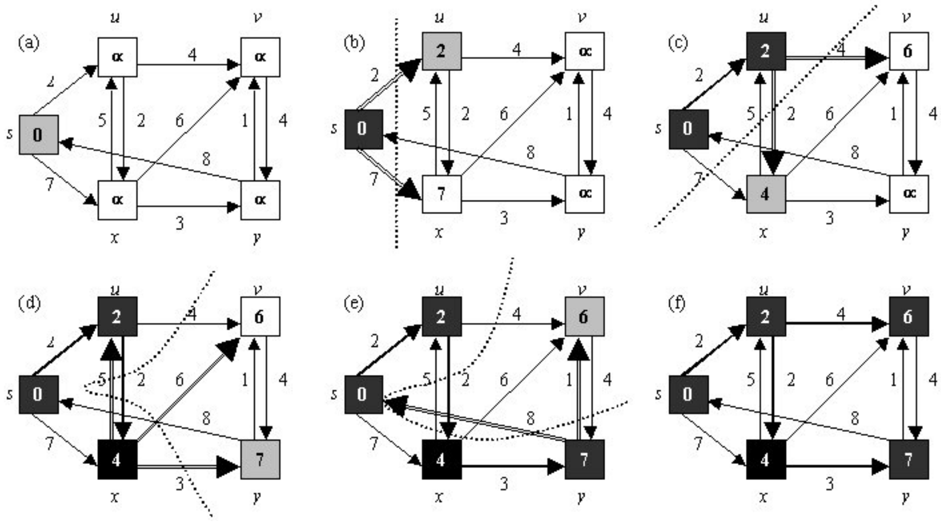


FIGURE 2.1: Dijkstra's algorithm path finding

As an example of how it finds the optimal path, let's use graph 2.1[1], the value inside the boxes is the distance that takes to go from the initial node (in this case the initial node is s) to each node. As seen in the pseudocode, all node values have been set to infinity except the departure node.

Nodes in gray represent the node that we are examining, nodes in black are those whose minimum distance has already been found, and the dotted line represents a virtual delimiter for the nodes whose distance has already been determined.

To find the shortest path from s to v, we start at s, since it has the minimum distance of all the nodes, and draw the virtual dotted line between it and the rest of the nodes (a and b). We now evaluate the vertex that are cut with that line, the lengths are 2

[1]Image from http://cs.smith.edu/~streinu/Teaching/Courses/274/Spring98/Projects/Philip/fp/dijkstra.htm

(between s and u) and 7 (between s and x). We now have evaluated the node u, so we update the position of the dotted line. We now consider all the nodes that cur that line and have not been considered yet, that cut gives us the values 4 (between u and v) and 6 (between x and v), since the minimum added distance from s to v (going through u) is less than infinity, we update its distance (c).

We keep this process until all distances have been found, or we can stop when the desired distance between the origin node and the destination node is found.

```
function Dijkstra(Graph, source):

    for each vertex v in Graph: // Initialization
        dist[v] := infinity // initial distance from source to vertex v
                            //is set to infinite
        previous[v] := undefined // Previous node in optimal path from source

    dist[source] := 0 // Distance from source to source

    Q := the set of all nodes in Graph // all nodes in the graph
                                        //are unoptimized - thus are in Q

    while Q is not empty: // main loop
        u := node in Q with smallest dist[ ]
        remove u from Q

        for each neighbor v of u: // where v has not yet been removed from Q.
            alt := dist[u] + dist_between(u, v)

            if alt < dist[v] // Relax (u,v)
                dist[v] := alt
                previous[v] := u

return previous[]
```

2.3.1.2 Bellman-Ford algorithm

Dijkstra's algorithm has a weakness with negative cycles, for example the solution for the graph in Figure 2.2, will not be the optimal, since the distance between A and C found by Dijkstra's will be 0 instead of the minimum of -200.

Bellman-Ford's algorithm is a well known variation of Dijkstra's algorithm that is able to solve graphs with in which some of the edges are negatives, as long as there is not a negative cycle. It is slower than Dijkstra's algorithm for the same problem but it is more versatile.

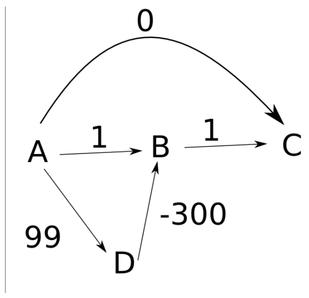


Figure 2.2: An unsolvable graph for Dijkstra

Its pseudocode is the following:

```
function BellmanFord(list vertices, list edges, vertex source)
  ::distance[],predecessor[]

  // This implementation takes in a graph, represented as
  // lists of vertices and edges, and fills two arrays
  // (distance and predecessor) with shortest-path
  // (less cost/distance/metric) information

  // Step 1: initialize graph
  for each vertex v in vertices:
      distance[v] := inf             // At the beginning , all vertices
                                     //have a weight of infinity
      predecessor[v] := null         // And a null predecessor

  distance[source] := 0              // Except for the Source,
                                     //where the Weight is zero

  // Step 2: relax edges repeatedly
  for i from 1 to size(vertices)-1:
      for each edge (u, v) with weight w in edges:
          if distance[u] + w < distance[v]:
              distance[v] := distance[u] + w
              predecessor[v] := u
```

```
// Step 3: check for negative-weight cycles
for each edge (u, v) with weight w in edges:
    if distance[u] + w < distance[v]:
        error "Graph contains a negative-weight cycle"
return distance[], predecessor[]
```

Given that there are not negative weights in our road network and that given the same graph, Bellman-Ford's takes longer than Dijkstra, this algorithm is not preferred to Dijkstra.

2.3.1.3 A* algorithm

A* is a variation of Dijkstra's algorithm widely used for pathfinding in videogames. Actually Dijsktra's algorithm is a special case of A*, when the heuristics are zero.

Its pseudocode is the following:

```
function A*(start, goal)
    // The set of nodes already evaluated.
    closedSet := {}
    // The set of currently discovered nodes still to be evaluated.
    // Initially, only the start node is known.
    openSet := {start}
    // For each node, which node it can most efficiently be reached from.
    // If a node can be reached from many nodes, cameFrom will eventually
    //contain the
    // most efficient previous step.
    cameFrom := the empty map

    // For each node, the cost of getting from the start node to that node.
    gScore := map with default value of Infinity
    // The cost of going from start to start is zero.
    gScore[start] := 0
    // For each node, the total cost of getting from the start node to the goal
    // by passing by that node. That value is partly known, partly heuristic.
    fScore := map with default value of Infinity
    // For the first node, that value is completely heuristic.
    fScore[start] := heuristic_cost_estimate(start, goal)
    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        closedSet.Add(current)
        for each neighbor of current
            if neighbor in closedSet
                continue // Ignore the neighbor which is already evaluated.
            // The distance from start to a neighbor
            tentative_gScore := gScore[current] + dist_between(current, neighbor)
            if neighbor not in openSet // Discover a new node
                openSet.Add(neighbor)
            else if tentative_gScore >= gScore[neighbor]
                continue // This is not a better path.

            // This path is the best until now. Record it!
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := gScore[neighbor] +
                                heuristic_cost_estimate(neighbor, goal)

    return failure

function reconstruct_path(cameFrom, current)
    total_path := [current]
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.append(current)
    return total_path
```

The main difference between A* and Dijsktra is that A* is an informed version of Dijkstra, where using the heuristic a priority can be set to chose one node over the other. It makes the pathfinding faster, but since getting a good heuristic requires a detailed study of the system, Dijkstra was chosen over A*,

2.3.2 Chosen algorithms

As detailed in the previous section, the most common algorithms have been studied to find the one that better fits the system. Each one has its benefits, but in the end Dijkstra's algorithm has been chosen due to its ease of use and modification.

2.3.2.1 Shortest path algorithm

The shortest path algorithm used in this simulator is the most basic implementation of Dijkstra's, knowing the distance between all the intersections the car finds its path looking for the one that takes less distance for it.

This method is deterministic, any car wanting to go from point A to point B will have the same path. Given the dynamic nature of the traffic, this will lead to significant traffic jams if the chosen path crosses a segment that congests very easily.

2.3.2.2 Fastest path algorithm

The fastest path algorithm is a slight modification of the previous algorithm, it only differs on searching for the minimum time instead of the minimum distance.

For each segment the maximum speed a car can drive on it is

$$Speed_{max} = min(\text{max speed on that segment}|\text{max speed of the car})$$

Because the maximum speed of each car is different, different paths from A to B can be found, which makes this algorithm slightly better to avoid traffic jams. However, it does so by accident and many cars will chose the same path.

2.3.2.3 Smart path algorithm

The smart path algorithm takes into account the state of the traffic in real time and avoids congested paths. This is the proposed algorithm and it is expected to behave much better than the others.

Again, this algorithm is a modification of Dijkstra's algorithm that minimizes the time that takes the car from going to point A to B. This algorithm is very similar to the previous one, but it takes into account the *current* maximum speed of that segment such that

$$Speed_{max} = min(\text{current max speed on that segment}|\text{max speed of the car})$$

so if a segment is congested the car simply avoids it. Because segments are dynamically congested, this algorithm recalculates its path at every intersection, given that you cannot turn around inside a segment this algorithm effectively keeps you in the fastest path at every moment.

Chapter 3

The multi agent system

This chapter gives an overview of which parts is the MAS composed, how its internal parts work, which communication schemes it uses and how it works.

JADE uses a peculiar system of agents and behaviors, agents represent a static class that keeps all the information stored but doesn't do anything by itself, that's the behaviors job. Behaviors are like scheduled actions for that specific class, such as moving the car this time unit or drawing the new added cars in the GUI.

It is important for behaviors not take a lot of time, since that is the way to sharing computing time between resources, you execute your behavior and *go to sleep* while other behaviors are executed. This makes for a really fast and efficient way of sharing resources if used correctly.

In this MAS, every behavior takes one Tick (it represent a second on the simulator timescale) to finish, then depending on its nature it will be repeated until an ending condition is reached.

3.1 Multi agent system models overview

In this section an overview of the different models that the system uses will be done, as well as a brief definition of the implemented classes.

3.1.1 Agent model

This subsection details the JADE agents used by the system.

3.1.1.1 Car agent

This agent represents a mobile vehicle with a set maximum speed that moves from a starting point to a destination via valid paths. The route that follows is determined by the type of routing algorithm chosen.

3.1.1.2 Event manager agent

This agent reads all the events from a file and executes them at specific points in time, this helps the system to have a deterministic behavior if anyone wants to redo the experiments.

3.1.1.3 Interface agent

This agent is keeps all the information about the GUI and is the one that has to keep all its information updated. It also reads the user inputs (such as changing the timescale).

3.1.1.4 Segment agent

This agent represents a road side unit on one segment of the network, that is the connection between two intersections. The road side unit covers the full extent of the segment. All car agents register on it when entering and deregister when leaving, he is in charge of communicating the GUI the position of its car agents, so that the number of messages is reduced (from one message per car to one message per segment).

3.1.1.5 Time keeper agent

This agent keeps the simulation synchronized between all agents, this makes keeping track of time and making scheduled events possible. This is the key to make all the cars move at the same time and, schedule events at a certain moment of the day (such as 12:00).

3.1.2 Behavior model

This subsection details the behaviors used by the system.

3.1.2.1 Car behavior

This behavior is used by the Car agent and calculates the next graphical position of the car. It also registers and deregisters the car from the segments. Registering is done when the car enters in a new segment and deregistering is done when the car exits a segment.

In every tick it moves and registers or deregisters of the segment if needed, this is a cyclic behavior that ends when the car reaches its destination.

3.1.2.2 Event manager behavior

This behavior is used by the Event manager agent. It executes the events that has in memory that have previously been read by its agent, if it is the time to execute that event it sends the instructions to the interested party. It also formats the time displayed in the GUI so that it is human readable.

In every tick it sends all the necessary messages if a event is executed, it also sends a message to update the time. This is a cyclic behavior that ends when the simulation ends.

3.1.2.3 Interface add car behavior

This behavior is used by the Interface agent, it receives the instructions to create a new car (from the manager behavior) and creates the graphical representation of it and keeps it updated.

In every tick it checks if a new car has to be added. This is a cyclic behavior that ends when the simulation ends.

3.1.2.4 Interface draw behavior

This behavior is used by the Interface agent (agents can have more than one behavior) and it updates all the parts of the GUI. It updates all the cars position on the GUI, adds new cars to the GUI, deletes cars that have finished from the GUI and updates the time and the number of cars on the GUI.

In every tick it checks whether he has to update the any part of the GUI or not (usually it has to update all the cars position since they move every tick). This is a cyclic behavior that ends when the simulation ends.

3.1.2.5 Segment listen behavior

This behavior is used by the Segment agent and listens to messages from cars to register or deregister, or to update their position. It also listens for messages from the Event manager in case it needs to change its service level.

In every tick it checks for messages from cars or events and updates its lists of registered cars. This is a cyclic behavior that ends when the simulation ends.

3.1.2.6 Segment send to draw behavior

This behavior is used by the Segment agent and sends the information about the cars that are registered on it to the Interface agent so it updates the car positions.

In every tick if there are any cars on it, it sends their information to the interface agent. This is a cyclic behavior that ends when the simulation ends.

3.1.3 Representation model

This subsection describes the classes that the system uses to model the road network.

3.1.3.1 Intersection

This class represents an intersection, that is a point on the map where one or more segments end or start. It is only possible for a car to change between segments here.

3.1.3.2 Segment

This class represents the connection between two intersections. Each segment has a origin and an end, it also keeps all the important information such as maximum speed, number of tracks and density. This class also has a list of steps that contain the information for its graphical representation.

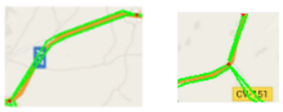


FIGURE 3.1: An example of segment and intersection

3.1.3.3 Step

This class is used to represent a line in the canvas where the graphical map is drawn, it contains the x and y coordinates for the initial and ending point of that line, segments are usually made up of more than one step so its graphical representation is real life like.

3.1.3.4 Path

This class is used to keep the segments and steps that a car has to follow to get from its origin to its destination, this path can be dynamically computed depending on the algorithm of choice.

3.1.3.5 Map

This class represents the whole road network, it reads the data from file and instances all the necessary classes to build the directed graph, it also instances the segment agents. It provides many methods to work with the graph.

3.1.4 CanvasWorld

This class contains all the information that is needed to draw the GUI, it also contains an API that allows the Interface agent to modify its contents.

3.1.5 Main

This is the class that is called when the program is started it spawns the required JADE services. It also provides a few configuration options that are worth mentioning:

- **tickLength**: This parameter is used when the application is used in headless mode, that is without spawning the GUI, which is very useful for servers. This overwrites the default tickLength that is usually set by the user via a slider in the GUI. *Default*: 1L
- **startingTick**: This is the tick at which the application starts, because this tick represents one simulation second you can do something like $7 * 3600 + 30 * 60$ to make it start at 7:30 am. *Default*: $7 * 3600 + 59 * 60$
- **finishingTick**: This is the tick at which the simulation will end. *Default*: $24*3600$

- **numberOfCars**: This parameter allows the user to put a certain number of smartcars at the beginning of the simulation, this is very useful for stress tests. *Default*: 0
- **drawGUI**: This parameter allows running the application in headless mode. *Default*: true
- **startRMA**: This parameter allows the control of starting the JADE Remote Agent Management, which is an interface to an administration panel for the agents. *Default*: false
- **segmentLogging**: This parameter allows the control of the logging system. *Default*: false
- **loggingDirectory**: This parameter allows us to change where the log files will be stored. *Default*: ""

3.1.6 Search algorithms

At the moment of writing this thesis, there are only three implemented algorithms. The system uses the Factory method programming pattern [18] so that if any researcher wants to add a new algorithm he or she can do it easily. The algorithms are further explained in Chapter 2.

3.1.6.1 Shortest path algorithm

This algorithms is an implementation of the Dijkstra's algorithm [19] that looks for the shortest path on the graph. Cars using this algorithm always take the same path given the same origin and the same destination.

3.1.6.2 Fastest path algorithm

This algorithm is a modification of Dijkstra's that focuses on time rather than on distance, this algorithm searches for the longest time for a given segment knowing the maximum speed of the car and of that segment. This **does not** take into account the state of the traffic.

3.1.6.3 Smartest path algorithm

This algorithm is yet another modification of Dijkstra's algorithm, this algorithm focuses on minimizing the trip time and takes into account the current traffic, so that if there is a congestion on a path and the average speed is slower, it will change to a longer path whose total trip time is lower.

3.1.7 Knowledge model

This subsection describes the files that make up the knowledge model database

3.1.7.1 Events

A file of events is needed so the simulator can run them, the file should be in *csv* format.

Type	**Time**	**Origin**	**Ending**	**Maximum speed**	**Algorithm**
newCar	08:53	I-CV10-08	I-N340-07	86	fastest
newCar	22:27	I-AP7-01	I-CV10-03	114	shortest
newCar	11:07	I-AP7-01	I-CV10-04	86	shortest
newCar	11:46	I-CV10-04	I-CS22-04	105	smartest

Table 3.1: A few rows of the events.csv file

A Python script to generate this file is also provided in the root of the project, called *generateRandomEvents.py* which allows for an easy creation of test files.

3.1.7.2 Map files

This are the files where the Map class takes its data:

- **intersections**: A JSON file containing the information about the intersections and their position on the graphical map.
- **segments**: A JSON file containing the information of the segments, including origin and destination, maximum speed, length, etc.
- **steps**: A JSON file that contains the steps that make up a segment, remember that the steps are the graphical representation of the segment, whereas the segment does not have a representation by itself.

3.1.7.3 Images

There are also three images that are needed on this simulator:

- **icon.png**: That's the icon that will appear on the top left corner of the GUI.
- **legend.png**: This is a legend of the simulator, it helps interpreting the data.
- **red.png**: This is the image used to represent the road network, a Google Maps screenshot.

3.2 Interaction model

Due to the distributed nature of this simulator, all the communication between classes has to be done through messages. JADE allows us to set different kind of communication ontologies so classes know what type of arguments to expect and what to do with those arguments. We can see in Figure 3.2 the different communication ontologies used between classes, this simulator uses quite a few. A brief description of them is done in Table 3.2.

3.3 Types of events

Events are a powerful aspect of the simulator, events change the simulation in real time. Currently there are only 2 types of events supported on the simulator, adding a new car with all its parameters such as the kind of routing algorithm that it will use, its maximum speed, its origin and its destination.

The second type of event is used to change the service level of a segment, this is very useful when trying to model the behavior of the agents in very specific or limit situations.

Events are usually read from file at the start of the simulation, and are executed by the Event manager according to its schedule. Any kind of event can be added to this file, with the time (on the simulation) that has to be executed on.

3.4 Service level of a segment

Depending on the density of a segment and its capacity, a quality of service is granted, this gives information about the conditions of the traffic, speed, travel time, safety, etc.

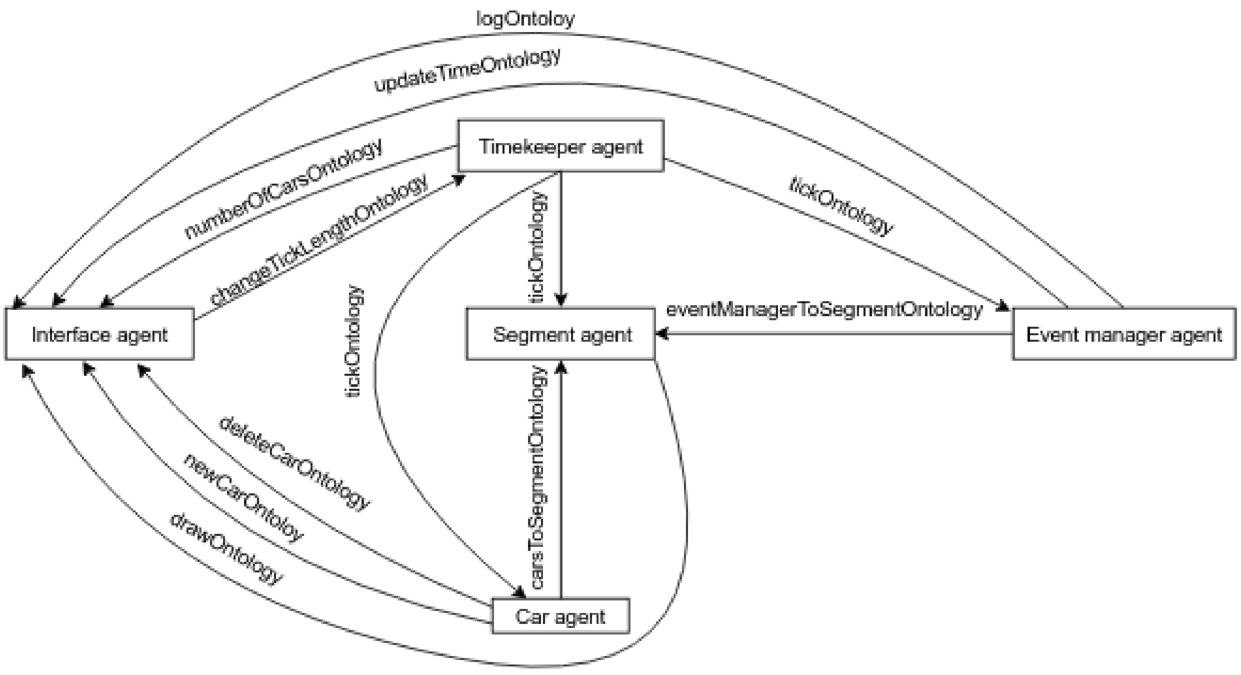


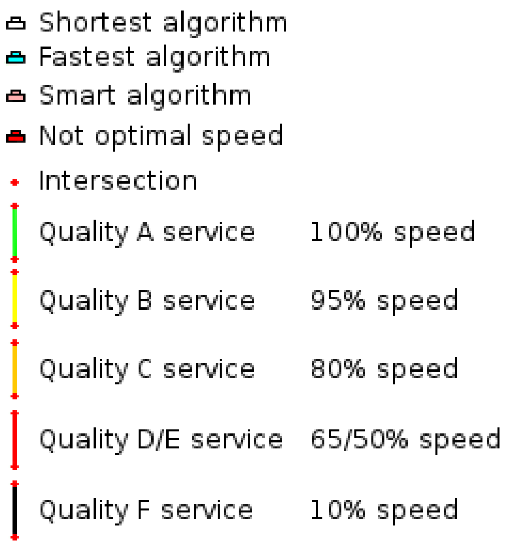
FIGURE 3.2: Ontology communications scheme

There are 6 main service levels according to Dirección General de Tráfico in Spain as can be seen in Table 3.3. The system also behaves appropriately on this sense and changes the service level dynamically. The GUI of the system allows to easily read on which service level each segment is.

3.5 Graphical user interface

When running a simulation in the MAS, it is helpful to have immediate feedback. As can be seen in the Figure 3.4, a real time map of the traffic is depicted. Figure 3.3, as mentioned before, shows an easily readable legend. Depending on which algorithm the agent is using, it is drawn with different colors, the same kind of feedback is obtained from the service level of the segments, they change their color to reflect their status.

The GUI also displays a clock in the upper right corner, that is the time *inside* the simulator, which in combination with the scrollpanel just bellow it that displays what



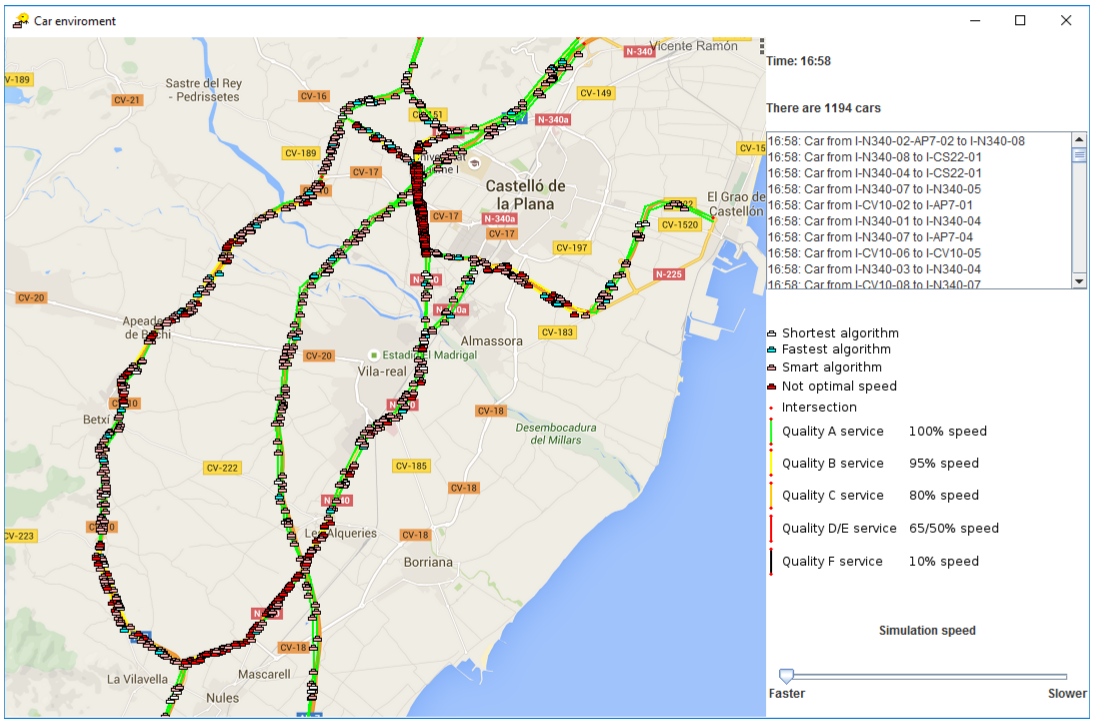
🚗 Shortest algorithm
🚗 Fastest algorithm
🚗 Smart algorithm
🚗 Not optimal speed
● Intersection

🟩	Quality A service	100% speed
🟨	Quality B service	95% speed
🟧	Quality C service	80% speed
🟥	Quality D/E service	65/50% speed
⬛	Quality F service	10% speed

FIGURE 3.3: Legend of the system

is happening, allows the researcher to keep track of all the events at any point of the simulation.

The last element of the GUI is a slider situated on the bottom right part, that allows the user to change the length between ticks (one tick represents one second inside the simulation), so special cases can be better studied.



Time: 16:58

There are 1194 cars

```
16:58: Car from I-N340-02-AP7-02 to I-N340-08
16:58: Car from I-N340-08 to I-CS22-01
16:58: Car from I-N340-04 to I-CS22-01
16:58: Car from I-N340-07 to I-N340-05
16:58: Car from I-CV10-02 to I-AP7-01
16:58: Car from I-N340-01 to I-N340-04
16:58: Car from I-N340-07 to I-AP7-04
16:58: Car from I-CV10-06 to I-CV10-05
16:58: Car from I-N340-03 to I-N340-04
16:58: Car from I-CV10-08 to I-N340-07
```

🚗 Shortest algorithm
🚗 Fastest algorithm
🚗 Smart algorithm
🚗 Not optimal speed
● Intersection

🟩	Quality A service	100% speed
🟨	Quality B service	95% speed
🟧	Quality C service	80% speed
🟥	Quality D/E service	65/50% speed
⬛	Quality F service	10% speed

FIGURE 3.4: Graphical user interface

Name	**From**	**To**	**Arguments**	**Description**
logOntology	Event manager agent	Interface agent	String	Tells the interface agent which text to add to the log panel
drawOntology	Segment agent	Interface agent	String	Sends the information about the cars that have to be drawn
newCarOntology	Car agent	Interface agent	String	Adds a car for the first time
changeTickLengthOntology	Interface agent	Timekeeper agent	Integer	Changes the ticklength on the simulation
tickOntology	Timekeeper agent	Car, segment and Manager agents	Long	Tells everyone to compute an additional tick
numberOfCarsOntology	Timekeeper agent	Interface agent	Integer	Tells the interface how many cars are running in the simulation
carToSegmentOntology	Car agent	Segment agent	String	Tells the segment to register, deregister or update this car
deleteCarOntology	Car agent	Interface agent	String	Tells the interface to delete that car
updateTimeOntology	Event manager agent	Interface agent	String	Tells the interface to update the displayed clock
eventManagerToSegmentOntology	Event manager agent	Segment agent	String	Tells a segment to change its service level

Table 3.2: The complete list of ontologies

Service level	**Description**
A	Free circulation, freedom of movement, high speed
B	Free circulation, less freedom of choosing track, maximum speed starts to be affected
C	Still stable circulation, now choosing the desired track is more of an imposition of the traffic than the willingness of the driver, slower speed
D	Instability, there is no freedom of choosing track, difficult keeping the same speed all the time
E	Instability, circulation starts to stop for brief periods, speed no faster than 50km/h
F	This state represents a traffic jam

Table 3.3: Service levels

Chapter 4

Tests and results

In this chapter the tests made in the simulator to test the behavior of the proposed algorithm will be explained, as well as the results.

4.1 Test setup

To test how the algorithm behaves in real life traffic scenarios, a 16 hour simulation will be done. 100000 cars will be randomly generated, their start and ending points can only be one of the 26 existing intersections, and they can chose any path combining the 58 available segments. The car maximum speed is between 80 and 120.

The test will be carried on 11 separated simulations, each of one containing a different percentage of cars using the proposed algorithm, from 0% to 100%. To get more interesting results, *the same cars* will be used in every simulation, that's the same origin, destiny, time of appearance, and maximum speed. The only thing that will change is that from those initial cars, a subset will be chosen and their routing algorithm will change to the smart algorithm.

The Figure 4.2 shows how the graph has been constructed, the weights are the distance in kilometers and the maximum speed between those two nodes. Also in Figure 4.1 we can see where the physical intersections are.

4.2 Results

The goal of this tests is to show how the smart cars behave in a dynamic environment. To see how the percentage of them affects the system, the data of all the segments have

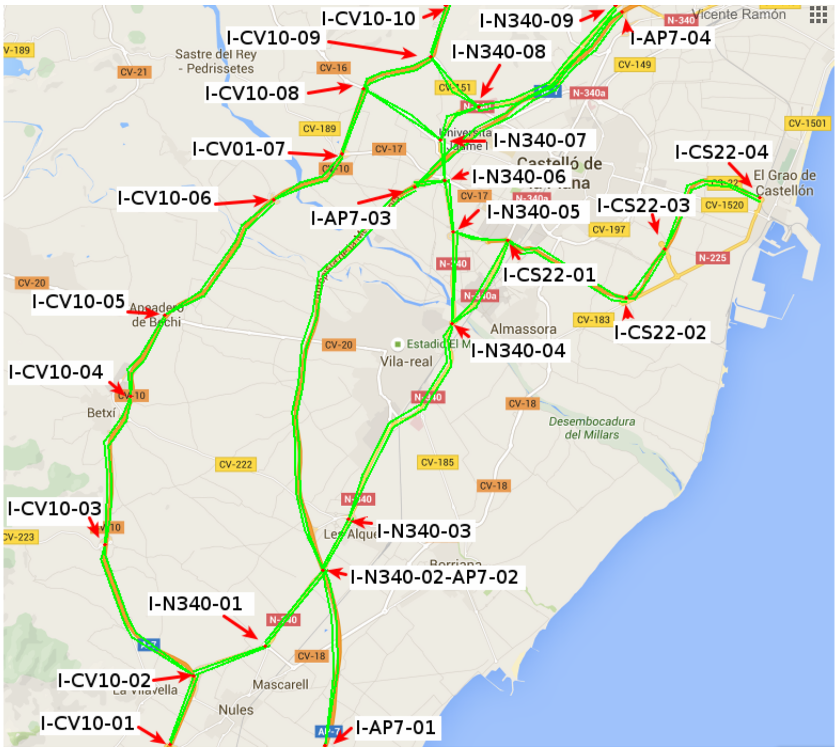


FIGURE 4.1: Road network map

been retrieved. This data contains the number of cars, the service level, the maximum speed for that segment and the current speed.

The current speed is dependent of the service level of the segment and is the maximum allowed speed at that point in time. Figure 4.5 (a bigger version of this figure can be found at Figure A.3) shows a plot for 16 hours for one of the most demanding segments. It is easy to see that the more cars there are in the segment, the slower the current speed. In this segment a extreme situation is never reached, but gives a good overview of how to used the output data from the MAS.

With this data, the best results we can take is the average current speed for all segments on a given simulation, and we can compare the 11 simulations to see how the system has performed. This will give us a very good idea of how the system becomes more stable and efficient the more smart cars there are. The results have been plotted in Figure 4.3 (a bigger version of this figure can be found at Figure A.1) and, as expected, the system behaves better when we increase the percentage of smartcars.

The peak at 8:00 is caused because prior to that there are no cars on the simulator, so they can move freely and at maximum speed at first. After that we can see a low speed area when the event manager starts adds all the agents, because all the agents are added in valid intersections and the system has not been stabilized some segments start to lower its service level. After that the system stabilizes with a low number of

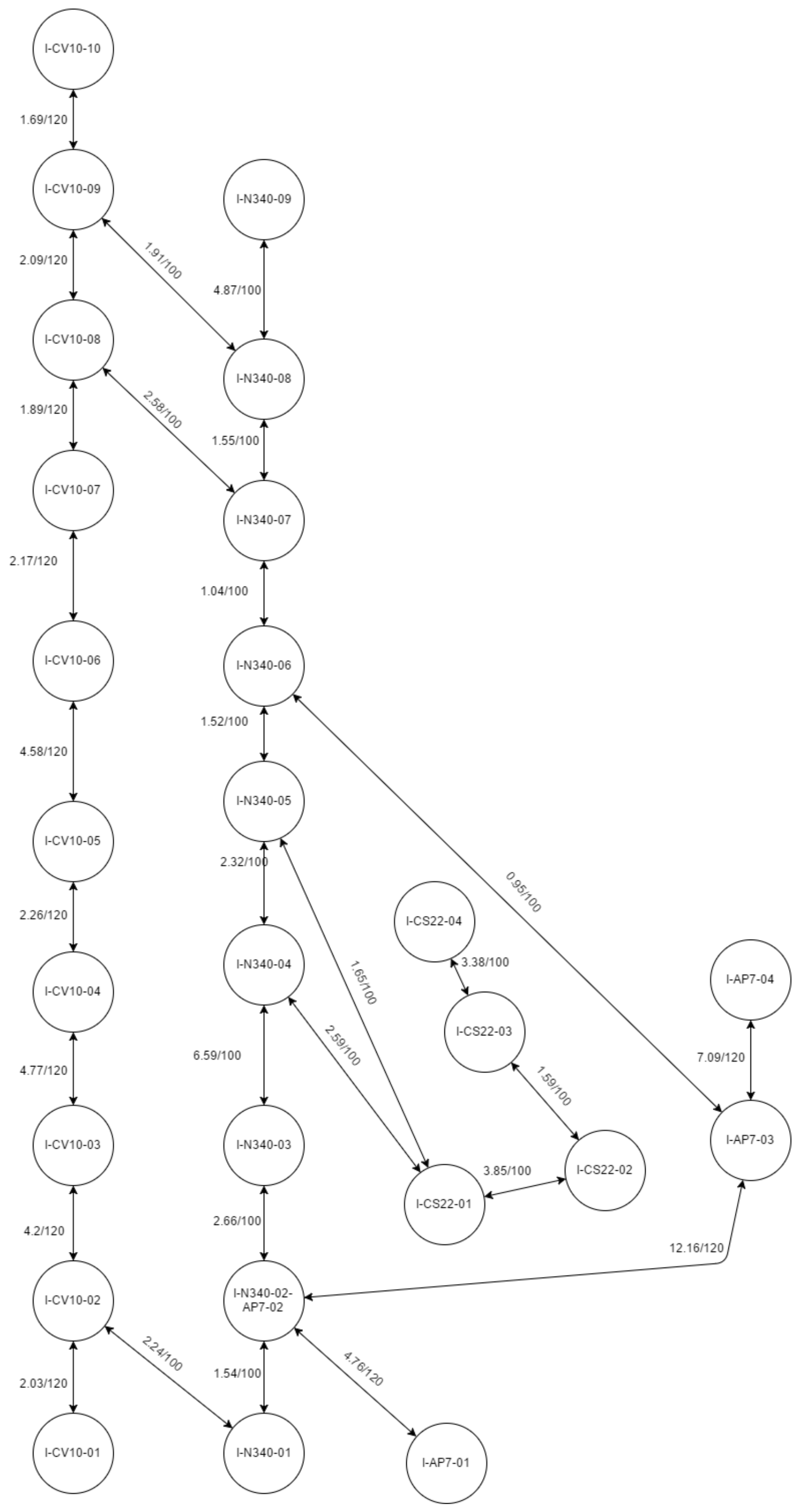


Figure 4.2: Road network graph

vehicles per segment, allowing full speed on most of them. And then, when cars are added the system fully stabilizes and smartcars start to dynamically adjust their routes.

The average number of cars per segment can be seen in Figure 4.4 (a bigger version of this figure can be found at Figure A.2) .

We can see that in both Figures, when the system does not have any smart node, it behaves really poorly. Its average speed as seen in Figure 4.3 is way lower than any other case. It is surprising that even with a 10% of the cars being aware of the traffic, the average speed has an improvement of 3 km/h. This is also obvious in the average number of cars per segment, Figure 4.4, when we can see that when there is a raise on the proportion of smartcars, there are less cars on the network. That is due to them reaching faster their destinations.

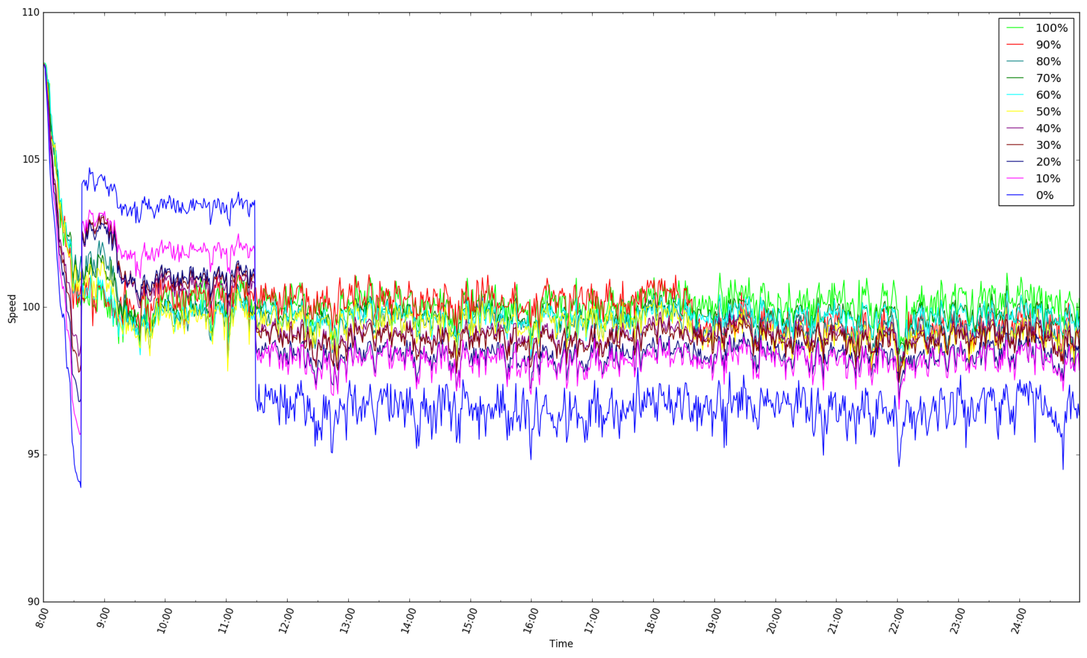


FIGURE 4.3: Results for the mean current velocity speed

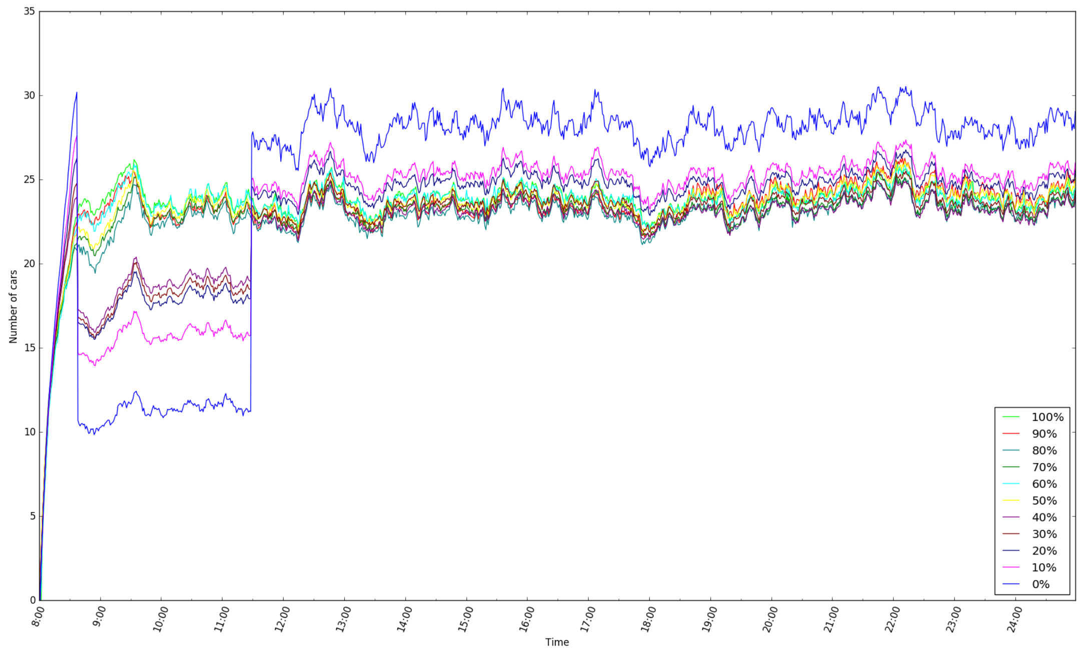


Figure 4.4: Results for the mean current number of cars

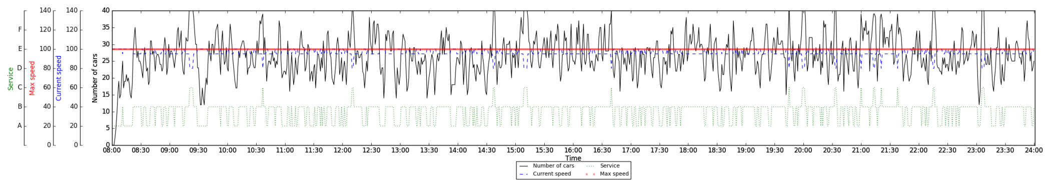


Figure 4.5: Traffic on the segment CS-22-01

Chapter 5

Conclusions and future work

5.1 Personal experience

This project was incredibly challenging, developing all the required models for the distributed multiple agent simulator took me longer than anticipated. This was also the most ambitious project I have ever developed, and I had to use every technique that I learned on the bachelor and on the master.

As mentioned in previous chapters, I used the JADE library and open source implementation of a distributed communication framework. While working with it I found two bugs that I had reported, and many times the only workaround until the fix came was simply changing the source code, recompiling and carrying on. The documentation on this library is very good, but for advanced MAS it was a little bit short.

I spent a big quantity of this projects time optimizing the system to support as many cars as possible, finding the limitations of this library at every corner. The biggest limitation that I found was how to deliver that many messages in a timely manner from the TimeKeeper agent to the rest of the agents to synchronize the simulation time, and that was my bottleneck for many weeks.

Also, since the GUI is updating itself very fast, I had to use all the tricks in the manual to make it as efficient as possible, and I had to learn how the most advanced swing techniques had to be applied.

Another big problem I had were the communication ontologies, I had to create unique ontologies so that the communication between agents could easily be extended.

The development of the routing algorithms was also challenging, but I did learn a lot about graphs and its representation, I even made my own system to store that graph in a human readable way to a file. The smart algorithm was very difficult at first, I tried and failed with many implementations of that algorithm that finally weren't good and simple enough.

In conclusion, I am incredibly proud of this project, it is modular, distributed, open source and I have employed many new technologies that I had never used before.

5.2 Conclusions

As has been shown in Chapter 4 even with a very little userbase, big gains can be achieved. This kind of technology not only helps those that use it, although they are the more benefited, but helps every user of the network. The traffic becomes more fluid, travel times shorten and, because cars spend less time on the road, the level of pollution decreases.

5.3 Future work

Ideally, in the future a real life scenario can be set to test the results of this thesis, with just a little bit of hardware this could be tested on a small section of the network.

Another model that could be tested on this MAS, now that is finished, is an algorithm that predicts which segments the cars that are in the network are going to take, and how that will affect the service level in the future. If the system were to know all the paths that all the cars want to take, further improvements could be achieved since no predictions would have to be made. But this last idea creates a privacy conflict and is why it has not been studied further.

Appendix A

Bigger figures

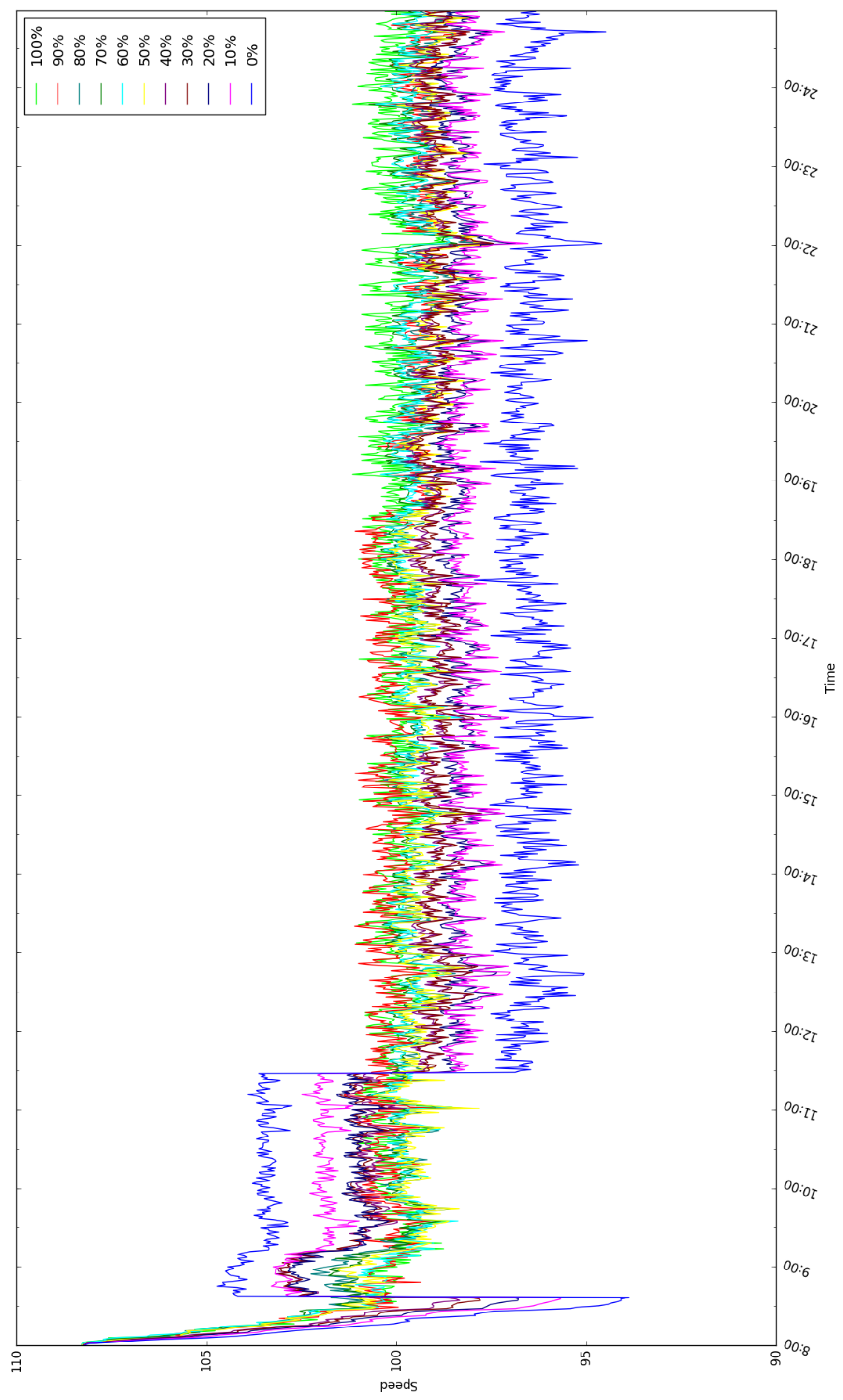


Figure A.1: Results for the mean current velocity speed

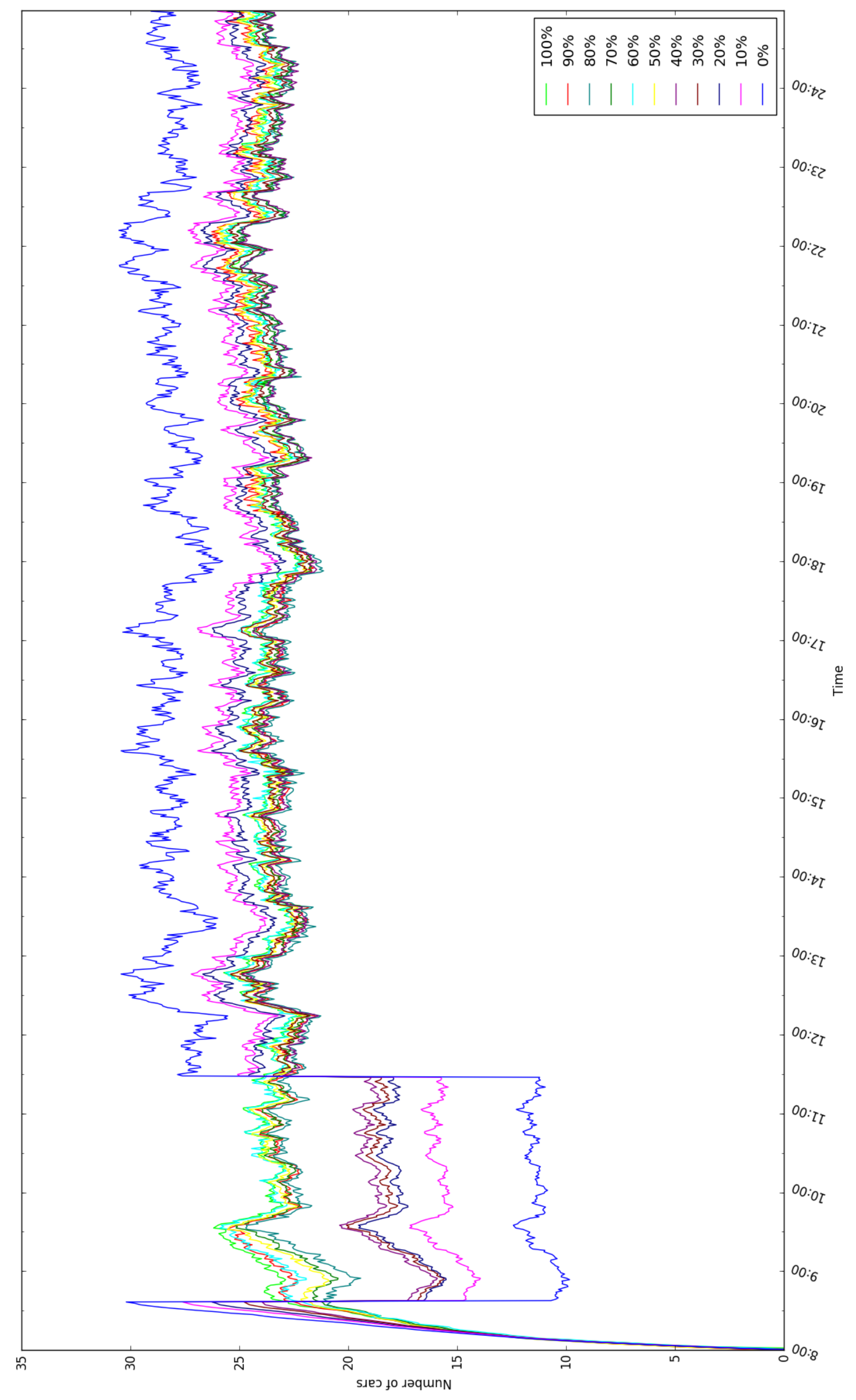


FIGURE A.2: Results for the mean current number of cars

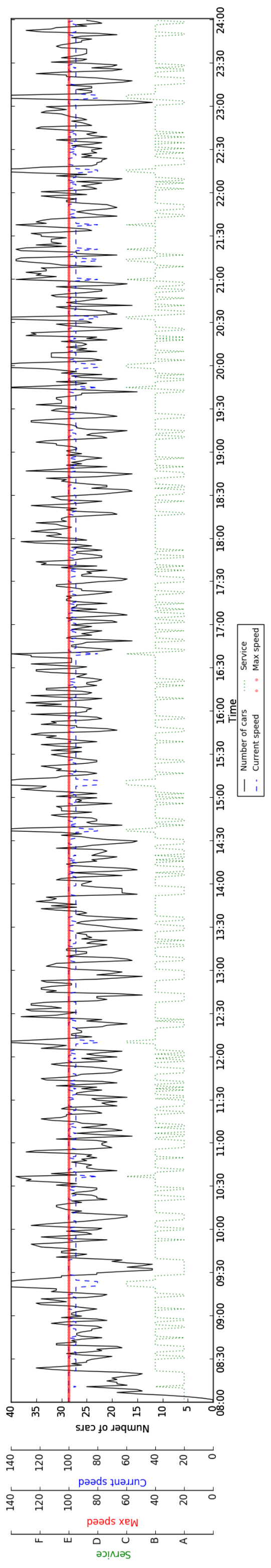


FIGURE A.3: Traffic on the segment CS-22-01

Bibliography

[1] Xue Yang, Jie Liu, Feng Zhao, and N.h. Vaidya. A vehicle-to-vehicle communication protocol for cooperative collision warning. *The First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004.* doi: 10.1109/mobiq.2004.1331717.

[2] Shane Tuohy, Martin Glavin, Ciaran Hughes, Edward Jones, Mohan Trivedi, and Liam Kilmartin. Intra-vehicle networks: A review. *IEEE Trans. Intell. Transport. Syst. IEEE Transactions on Intelligent Transportation Systems*, 16(2):534–545, 2015. doi: 10.1109/tits.2014.2320605.

[3] Noam Nisan. *Algorithmic game theory.* Cambridge University Press, 2007.

[4] By Java. Java software. URL https://www.oracle.com/java/index.html.

[5] Write once, run anywhere, . URL https://en.wikipedia.org/wiki/write_once,_run_anywhere.

[6] Package javax.swing. URL https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html.

[7] URL http://jade.tilab.com/.

[8] URL http://www.fipa.org/.

[9] Andrea Caragliu, Chiara Del Bo, and Peter Nijkamp. Smart cities in europe. *Journal of Urban Technology*, 18(2):65–82, 2011. doi: 10.1080/10630732.2011.601117.

[10] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of things for smart cities. *IEEE Internet of Things Journal*, 1(1): 22–32, 2014. doi: 10.1109/jiot.2014.2306328.

[11] Anthony M. Townsend. *Smart cities: big data, civic hackers, and the quest for a new utopia.* W.W. Norton and Company, 2013.

[12] Jesse Shapiro. Smart cities: Quality of life, productivity, and the growth effects of human capital. *MIT Press journals*, 2005. doi: 10.3386/w11615.

[13] Smart city, . URL https://en.wikipedia.org/wiki/smart_city.

[14] Dick Lenior, Wiel Janssen, Mark Neerincx, and Kirsten Schreibers. Human-factors engineering for smart transport: Decision support for car drivers and train traffic controllers. *Applied Ergonomics*, 37(4):479–490, 2006. doi: 10.1016/j.apergo.2006.04.021.

[15] Johanna Camargo Pérez, Martha Helena Carrillo, and Jairo R. Montoya-Torres. Multi-criteria approaches for urban passenger transport systems: a literature review. *Annals of Operations Research Ann Oper Res*, 226(1):69–87, 2014. doi: 10.1007/s10479-014-1681-8.

[16] Muhammad Adeel Javaid. Understanding dijkstra algorithm. *SSRN Electronic Journal*. doi: 10.2139/ssrn.2340905.

[17] Steven S. Skiena. Weighted graph algorithms. *The Algorithm Design Manual*, page 191–229, 2012. doi: 10.1007/978-1-84800-070-4_6.

[18] Factory method pattern, . URL https://en.wikipedia.org/wiki/factory_method_pattern.

[19] Dijkstra's algorithm, . URL https://en.wikipedia.org/wiki/dijkstra's_algorithm.