UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN INFORMACIJSKE TEHNOLOGIJE

ZAKLJUČNA NALOGA
(FINAL PROJECT PAPER)

RAZVOJ SIMULATORJA NA PODLAGI AGENTOV ZA OPTIMIZACIJO TOPOLOGIJE PROMETNEGA OMREŽJA Z UPORABO GENSKEGA ALGORITMA (DEVELOPING AN AGENT-BASED SIMULATOR FOR OPTIMISING TRAFFIC NETWORK TOPOLOGY BY APPLYING A GENETIC ALGORITHM)

PATRIK MATOŠEVIĆ

MATOŠEVIĆ ZAKLJUČNA NALOGA (FINAL PROJECT PAPER) 2022

UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN
INFORMACIJSKE TEHNOLOGIJE

Zaključna naloga
(Final project paper)

Razvoj simulatorja na podlagi agentov za optimizacijo topologije prometnega omrežja z uporabo genskega algoritma

(Developing an agent-based simulator for optimising traffic network topology by applying a genetic algorithm)

Ime in priimek: Patrik Matošević
Študijski program: Računalništvo in informatika
Mentor: izr. prof. dr. Jernej Vičič
Somentor: asist. Aleksandar Tošić

Koper, september 2022

Ključna dokumentacijska informacija

Ime in PRIIMEK: Patrik MATOŠEVIĆ

Naslov zaključne naloge: Razvoj simulatorja na podlagi agentov za optimizacijo topologije prometnega omrežja z uporabo genskega algoritma

Kraj: Koper

Leto: 2022

Število listov: 40 Število slik: 4

Število referenc: 19

Mentor: izr. prof. dr. Jernej Vičič

Somentor: asist. Aleksandar Tošić

Ključne besede: Simulator, Genski algoritem

Izvleček:
V diplomskem delu se bomo osredotočili na razvoj simulatorja prometa na podlagi agentov in hevristični algoritem, namenjen optimizaciji konfiguracije vrst križišč. Simulator bo služil kot fitnes funkcija hevristike. Naš pristop bomo potrdili z optimizacijo cestnega omrežja Kopra v Sloveniji.

Key words documentation

Name and SURNAME: Patrik MATOŠEVIĆ

Title of final project paper: Developing an agent-based simulator for optimising traffic network topology by applying a genetic algorithm

Place: Koper

Year: 2022

Number of pages: 40 Number of figures: 4

Number of references: 19

Mentor: Assoc. Prof. Jernej Vičič, PhD

Co-Mentor: Assist. Aleksandar Tošić

Keywords: Simulator, Genetic algorithm

Abstract:
In the thesis we will focus on developing a simulation framework featuring an agent based traffic simulator, and a heuristic algorithm aimed at optimising the configuration of intersection types. The simulator will serve as the fitness function of the heuristic. We will validate our approach by optimising the road network of Koper, Slovenia.

Acknowledgement

I would like to thank the mentor doc. dr. Jernej Vičič and co-mentor mag. Aleksandar Tošić for all the help and support in preparing the final work and through out the years of faculty.

I would also like to thank all the professors from the Faculty of Mathematics, Natural Sciences and Information Technology for everything they taught me.

Special thanks goes to my parents and family who supported and enabled me to focus on my studies.
I would also like to thank all my friends who stood by me during my studies. Especially my friend Petar Deljanin who was part of the project which lead to this final work.

Contents

1 Introduction **1**

2 Literature Review **2**

2.1 SUMO [3] 2
2.1.1 Abstract 2
2.1.2 Introduction 2
2.1.3 Network Editor 2
2.1.4 Conclusion 3
2.2 AIMSUN [12] 3
2.2.1 Introduction 3
2.2.2 About 3
2.2.3 Conclusion 3
2.3 CORSIM [12] 4
2.3.1 Introduction 4
2.3.2 About 4
2.4 Comparison and Conclusion 4

3 Types Of Trafic Simulators **5**

3.1 Overview of traffic simulation 5
3.1.1 Introduction 5
3.1.2 Traffic models 5
3.1.3 System planning 6
3.1.4 Transport engineering 8
3.1.5 Software 9
3.2 Comparison of continuous and discrete time simulations 9
3.3 Continuous or Real time simulators 10
3.3.1 Simulation software 11
3.3.2 Modern applications 11
3.4 Discrete time 11
3.5 Agent based 12

4 Implementation **14**
4.1 Architecture . . . 14
4.2 Actor . . . 15
4.3 Simulation . . . 16
4.4 Vehicle . . . 16
4.5 Network . . . 17
4.5.1 Dijkstra algorithm . . . 18
4.5.2 Emit functionality . . . 20
4.6 Intersections . . . 20
4.6.1 Regular intersection . . . 20
4.6.2 Semaphore . . . 23
4.6.3 Roundabout . . . 23
4.7 Genetic Algorithm . . . 23
4.8 Results . . . 28

5 Conclusion **30**

6 Bibliography **31**

Table of Figures

1 The table defines different type of simulations and how they are devided by time and space. 7
2 This picture shows the GUI of the simulator which represents the map of Koper and all of the nodes and connections on which the traffic flows. 15
3 This diagram shortly describes a flow of a basic genetic algorithm so it can be visualized 27
4 The first run of the Simulator with GA applied 28

List of Abbreviations

SUMO - Simulation of Urban Mobility

TSS - Transport Simulation

FHWA - The Federal Highway Administration

GA - Genetic algorithm

GUI - Graphical user interface

DES - Discrete event simulation

IBM - Individual-Based Model

1 Introduction

Traffic congestion in urban areas is an ever increasing problem, because of the growth of number of cars that are owned and driven daily. There is a lot of research being done in this field and a lot of different attempts of solutions [10]. To test all of the theoretical solutions there needs to be some kind of a way to model them. This task is done by the urban traffic simulators, which can be helpful in providing a way of modeling this solutions. A simulator which provides a way to manipulate the environment would provide a great platform for testing algorithms that could lead to a better solution.
The aim of this thesis is to build an agent based simulator which is able to change the network topology on the fly. The simulator will then be used as a fitness function by selected heuristic algorithm to explore ways of optimising urban traffic topology and decreasing network congestion.
The thesis is divided into: Literature review which present existing simulators and their pros and cons, followed by a breakdown of types of traffic simulators which will make the reader more knowledgeable about different types of traffic simulators that already exist, next on the list is the implementation of our simulator with the presentation of the attempt of providing a simulator which can be used in a simple but effective way by the heuristic algorithms, the reader will get to know the architecture and the steps of implementation, after that we will go over the genetic algorithm used to explore a possible way of optimisation of the given topology. Finally we will show examples of the usage of the simulator and show the end results.

2 Literature Review

This section is dedicated to go over some of the bigger traffic simulators that already exist. Such as SUMO [3], AIMSUN [12] and CORSIM [12]. We will focus on description of each simulator and what are their pros and cons.

2.1 SUMO [3]

2.1.1 Abstract

SUMOLib is an open source library for the simulation of transportation networks. It includes modules for network generation, routing, demand estimation and traffic assignment. It also provides interfaces for visualization and analysis tools. The main focus lies on the development of a modular framework that allows users to easily add new features.

2.1.2 Introduction

SUMO is an open source project that began in 2001 by The German Aerospace Center. Its goal is to create a complete toolset for simulating traffic flows on networks. It includes a simulator, a visualization system, and a remote control interface. It also supports a large number of file formats. There is a high performance simulation which can be used on single junctions or whole cities. The simulation can also be controled by a "TracCI" interface which alowes online adaptation of the simulation.

2.1.3 Network Editor

Since 2011, a graphical network editing tool has been implemented. It allows you to add all necessary information about lanes, speeds, junctions and traffic lights to your simulation. Currently this tool is not part or included in any open source version, but it is still available for internal use only.

2.1.4 Conclusion

As we could see from the information above the real problem with using SUMO is the lack of the ability to easily change the network on the fly. All of the changes can be neatly done by the graphic UI, which is not useful for the application of genetic algorithms. GA require a lot of network changes and adaptations which is not supported or is very difficult with the SUMO traffic simulator.

2.2 AIMSUN [12]

2.2.1 Introduction

AIMSUM is an available TSS-Transport Simulation System which is being designed in Spain. The simulator is able to reproduce the real conditions of traffic of any network transport. The system is used for testing purposes of traffic management rules, access controls, traffic control systems, public transportation networks, lanes, and can also be combined with vehicle guidance systems, and other real-time applications. One of the standout features of AIMSUM is the ability to model different network models in the same simulation.

2.2.2 About

GETRAM [6] is a simulation platform designed for users of all levels. Users can create simulations using a drag and drop interface, edit existing ones or run them within the environment. AIMSUM is embedded within GETRAM, allowing users to simulate and analyze transportation networks. Users can also interact with other tools available through GETRAM, which are the TEDI [1] traffic network editor, network database, a model for performing simulation and an Application Programming Interface.

2.2.3 Conclusion

Although AIMSUM upon first inspection looked very promising, it proved to be still very hard to manipulate and also the MITSIMLab [4] which is the open source application that represents the actual world in AIMSUM is written in C++. Also after doing the research there is a student edition of the software which does not have all of the features.

2.3 CORSIM [12]

2.3.1 Introduction

CORSIM is a microscopic network simulator that integrates the NETSIM and FRESIM models. It is a very sophisticated and advanced traffic simulation tool that simulates traffic flow on integrated networks. It is capable of simulating both free flow and congested conditions. It is also capable of simulating multiple types of vehicles including cars, trucks, buses, motorcycles, bicycles, etc.

2.3.2 About

CORSIM, sponsored and developed by The Federal Highway Administration (FHWA), is a traffic simulation software that is used to model signal systems, road networks, and highway systems. It combines to models NETSIM which is for traffic on city roads, and FRESIM that represents traffic on roads and highways. CORISM has it's own interface and driver software which provide access to a new output data processor. The processor gives users the ability to collect specifically selected statistics and summary data during multiple simulation runs. The data is then written to an Excel workbook, comma separated or tab separated file.

2.4 Comparison and Conclusion

After an overview of different software we can saw that only AIMSUM can simulate three models at the same time while other were only microscopic simulators. SUMO and AIMSUM simulate traffic in a continuous time, while CORSIM uses a discrete system.

Simulator	Model	Type	API	Infrastructure	
				Difficulty	Flexibility
AIMSUM	all	continuous	non-existent	difficult	flexible
CORISM	microscopic	discrete	simple	medium	very limited
SUMO	microscopic	continuous	difficult	difficult	very limited
OUR	microscopic	discrete	simple	easy	flexible

All of the listed simulators require difficult or heavy coding for network manipulation which is the main goal of this thesis, for this reason we concluded that the best thing to do is develop our own actor based discrete time simulator which would be used to easily manipulate the network by applying genetic algorithm.

3 Types Of Trafic Simulators

3.1 Overview of traffic simulation

3.1.1 Introduction

Simulation of traffic is a method of simulating transportation systems with the use of mathematical modeling which include grid systems, freeway junctions, roundabouts, etc. To better help project, scheme, and operate transportation systems, we apply computer software. The first transportation systems started over forty years ago and is still a very important part in traffic engineering and transportation planning today. Numerous transport agencies, academic institutions and other traffic focused organisations use simulation and simulating software to aid their research and management of traffic networks.
Transportation simulation is an essential tool for studying complex systems like traffic flow, air pollution, energy consumption, and other aspects of urban life. It is also useful for testing ideas about improving transportation, developing policies, and planning cities. Simulation allows us to test our theories and hypotheses before we invest time and money in real experiments [2].
Simulation, in the scope of this thesis, is a mathematical representation of a physical system that allows the user to study the system's behavior. A simulation is usually represented by a graph of states that represent the different possible configurations of the system at each point in time. These graphs are called state diagrams. Simulations are often used to test hypotheses about systems before actually implementing them. For example, we might simulate a traffic flow model to see if our proposed changes will cause congestion.

3.1.2 Traffic models

Simulation models in transportation can use many different types of theories. Some of them are based more on the field of mathematics which is out of scope of this thesis but below I will provide some examples and go shortly over some of the traffic models that are present over time [19].
One of the first discrete event simulations models is the Monte Carlo simulation [7],

where a sequence of randomized numbers is used to incorporate traffic conditions.
Monte Carlo simulations are often used to simulate complex systems. For example, they are commonly used to model the spread of diseases, financial markets, weather patterns, and even the movement of particles through matter. A simple example of a Monte Carlo simulation is rolling dice. If you roll two sixes, then you win $100. However, if you roll a three, then you lose your money. You could also use a Monte Carlo simulation to simulate the movement of particles through a gas. If you put a particle in a box, and let it move around randomly, then you will eventually get a pattern of particles that resembles a cloud.
The Monte Carlo [7] method was followed by the Cellurar automata model [8] that uses deterministic rules to generate randomness.

Most of the recent methods use either discrete event simulation or continuous-time simulation. Discrete Event Simulation Models are stochastic (random) and dynamic (time changes). A queuing system is an example of a discrete event simulation model. Queues are often located at a single point in space and time. For instance, traffic lights are often placed at intersections.
In continuous time simulation, the system is simulated continuously. There is no need to define an initial condition, nor any requirement to specify the end point of the simulation. Continuous time simulation is often used when there is no fixed time step, such as in fluid dynamics simulations. The method behooves the use of differential equations(numerical integration methods).

A class called car-following models are based on differential equations. These models describe the motion of vehicles traveling in a line behind each other. A car following another car will adjust its speed to match the speed of the other vehicle.
This model helps us understand how cars behave when there is an accident and how they react to other vehicles around them. We can also study how they move through intersections and what happens at stop signs. Traffic engineers use this model to help plan roadways and determine the best locations for traffic lights.

3.1.3 System planning

These methods that we talked about are used to automate the development of a system, and are usually focused around specific areas of interests under a range of conditions. For example, a method might be used to test the impact of changing the number of lanes on the capacity of a road.
Traffic planning and forecasting can help us understand what we need to plan for, and

Types Of Simulation In Transportation

Time	State	Space		
		Continuous	Discrete	N/A
Continuous	Disc.	Real Transportation Systems * Traffic flow, pedestrians Dynamic traffic assignment		Discrete Event Systems * queueing inventory manufacturing
Continuous	Cont.	PDE Traffic flow models Pedestrian models		ODE vehicle motion car suspension queueing (fluid approx)
Discrete	Disc.		Cellular Automata * Traffic, pedestrians Land use Urban sprawl Random Number Generation	Discrete Event Simulation * queueing inventory manufacturing
Discrete	Cont.	Car-following models * Microscopic traffic flow models *	Numerical PDE methods Godunov, Variational	Numerical ODE methods Euler, Runga-Kutta time-series * ARIMA
N/A	Disc. or Cont.	Monte Carlo method * : use of pseudo-random number Simulation of static probabilistic problems Integration, Optimization		Econometric models trip generation, distribution, modal split Optimization static traffic assignment

Figure 1: The table defines different type of simulations and how they are devided by time and space.

what our goals should be. We can also see if there are any bottlenecks in the system, and what changes we might need to make to get around them. Traffic forecasting helps us predict when certain roads will become congested, and how long it will take for traffic to clear.

3.1.4 Transport engineering

Traffic simulation models can be useful in a range from microscopic, macroscopic, mesoscopic and sometimes even nanoscopic viewpoints. Transportation simulation models are used to plan and operate transportation systems. These simulations can help us understand how different policies affect traffic congestion, travel times, fuel consumption, emissions, etc. Simulation models can also be used to test new ideas about transportation policy or technology before implementation. Regional Planning Organizations use these models to evaluate different policy options. Air Quality Models help plan for future emissions and pollution levels. Transportation System Operations Modeling focuses on the operation of a single mode of transport, like walking or driving. Pinch Point Modeling helps determine the best location for traffic signals or stop signs. Lane type, signal timing and other transportation related questions are investigated to understand how to improve local system effectiveness. While certain modeling tools are specialized to model either operational or planning aspects of transportation, certain tools have the ability to model both to some extent. Whether it is for operations or planning, simulations can be used to model a variety of transportation modes including rail, bus, trucking, aviation, maritime, etc.

Macroscopic simulations

Macroscopic simulations target traffic flow modelling by using a high-level mathematical models. This kind of simulation is useful when you need to simulate large areas of land at once. For example, if you wanted to see what would happen if all cars were suddenly banned from driving on your street. Macroscopic simulations are also very quick and require less computing power because they don't require any detailed models of individual vehicles [9].

Microscopic simulations

Microscopic simulations put in front modelling of individual entities with high level of detail. A possible entity could be a traveller, vehicle, traffic light, etc. This type is often used for the study of urban traffic. It allows you to analyse both macroscopical and microscopic aspects (e g, traffic lights algorithm, multimode traffic) of the system. Microscopic simulations may also lead to longer computation time [9].

Mesoscopic simulations

Mesoscopic simulations are a mix between above mentioned simulation models. Traffic agents are modelled at a high level of detail, but the interactions and behaviours of the traffic agents appear to be less detailed than macroscopic approaches [9].

Nanoscopic simulations

Nanoscopic simulations are the most detailed of the mentioned approaches. This kind of simulations are used for testing and modelling autonomous driving. Autonomous driving is an emerging technology that allows cars to drive themselves. Internal systems like steering, braking, acceleration, and even navigation are simulated using software. These simulations allow developers to test the system before implementing them in real life [9].

3.1.5 Software

Simulation software is established upon modeling a real situation with the help of mathematical formulas. Simulation software is used to test products before they're built. It helps engineers see what changes need to be made to ensure that the finished product meets specifications. Real time simulations are used in many industries. For example, when an airline pilot needs to practice landing an aircraft, he or she will connect a mock cockpit to a simulator program that mimics the physical responses of the plane. This allows them to practice safely before actually flying the real thing. Simulation programs are also used in nuclear power plants to train workers on safety procedures. If there is ever a problem, the simulation program can be connected to a real-life version of the control panel, allowing the worker to see what happens if something goes wrong. Simulation Software is getting better in a number of different ways. New advances in Mathematics, Engineering and Computing are making Simulation Software Programs increasingly faster, more powerful, detailed and realistic. Transportation models generally can be separated into microscopic, mesoscopic, macroscopic, and metascopic models as mentioned and described above. A lot of software programs have a problem with external adjustment of network and are not very tilted to external changes still.

3.2 Comparison of continuous and discrete time simulations

In mathematics, a dynamical system is a mathematical model of a physical system whose states evolve according to a rule. For example, an ideal gas is a dynamical

system because its volume changes depending on temperature. However, a gas is not a discrete dynamical system since the volume does not jump. Instead, the volume changes continuously. Discrete dynamical systems are modeled using discrete time models, whereas continuous ones are modeled using differential equations [15].
In continuous systems, the state variables change continuously over time. For example, the position of an object changes continuously over time. This continuous change of the state variable is described by a differential equation. Continuous simulation is possible because we can represent any number exactly [15].
In discrete event simulation, state variables change only when an event happens. For example, if you were simulating a traffic light, the state variable could be whether the light is red, yellow, green, or flashing. Discrete event simulation is often used to simulate complex systems like factories, transportation networks, and supply chains.
Continuous dynamic systems can be modeled by continuous simulation models, while discrete dynamic systems require discrete event simulation models. Birth, death and predator- prey interactions are examples of discrete events. Continuous models can describe the dynamics of populations, whereas discrete models can be used to simulate individual behaviors. For example, if you want to model the growth of a population of bacteria, then a continuous model might be appropriate. However, if you want to understand the behavior of an individual bacterium, then a discrete model might be more useful [15].

3.3 Continuous or Real time simulators

As said before continuous time simulation is based on a set of differential equations. Continuous simulation is a type of simulation that involves a continuous flow of information. The equations are continuously updated and changed by the changing environment. Continuous simulations allow us to understand the behavior of systems that cannot be studied using discrete models. For example, we can simulate the movement of a ball in space, the growth of plants, or the evolution of an ecosystem.
A conceptual model represents the system on an abstract, idealised level. In order to create the conceptual model, two different approaches exist: The deductive method: The behaviour of the systems arises from physical laws that apply to all systems. The inductive method: The behaviour emerges from observing the behaviour of a single example. An example for a continuous simulation is the predator/prey model [11] [15].
Continuous Simulation is a modeling technique that allows you to simulate a real life event. You can see your simulated events in a timeline, just like in reality. Continuous Simulation is also commonly referred to as "real time" simulation because you can view the results of your simulations in real time. The main advantage of continuous

simulation is that you can easily change variables and parameters while you're running your simulation. For example, if you want to test out different scenarios, you can simply change the variables and run another simulation. Continuous Simulation is useful for many industries including aerospace, automotive, manufacturing, energy, and healthcare.

3.3.1 Simulation software

Graphical programming tools like VisSim and SimCAD Pro let users create continuous models quickly and easily. These programs integrate well with other systems, allowing users to simulate multiple processes at once. They also give users the ability to conditionally execute subsystems, speeding up the simulation time while preventing numerical errors. Graphical modeling software can be used as a training tool to help managers and operators understand complex systems [18].

3.3.2 Modern applications

Continuous simulation is found:

- inside Wii stations
- commercial flight simulators
- jet plane auto pilots
- advanced engineering design tools

Indeed, much of modern technology that we enjoy today would not be possible without continuous simulation [15].

3.4 Discrete time

A discrete event simulation models a system as a series of discrete events happening at specific times. If you think about something like an airplane flight, each step of the plane flying through the air is modeled as a discrete event. A DES is useful when modeling complex systems that have multiple interacting components. For example, a car manufacturer might create a DES of a vehicle manufacturing line to help them understand what happens if there is a problem with a component.
Fixed-increment time progression is an alternative to next-event time simulation. It breaks up time into smaller increments and simulates each increment independently.

Since not all time slices need to be simulated, fixed-increment time simulations can often run much faster than next-event time simulations.
In both forms of DES, the state changes discontinuously at discrete times. However, in continuous simulation, the state changes continuously over time on the base of a set of differential equation describing the rate of change of state variables, while in discrete event simulation, the state changes discontinously at discrete times.

To learn how to build discrete-event simulations we use queues, such as example of a bank(customers and tellers). In this example, the system entity is the customer, the system event is the arrival of the customer, the system state is the number of customers in the queue, and the system behavior is the service time of the teller. A discrete event simulation is a type of simulation that models the interaction of agents, where each agent represents a single object in the real world. For example, if there are 10 customers waiting to be served, then there will be 10 agents representing those customers. Each agent will have its own attributes like age, gender, and income. The system events are when the customers arrive and depart. When an agent arrives, the system state changes to represent how many customers are in the queue. If a customer departs, the system state changes again to reflect the fact that fewer customers are in the queue now. Finally, the system behavior is the amount of time it takes to serve each customer. Agents may also have different behaviors depending on what happens during the simulation. For instance, if a customer is late, the agent might become angry and start cursing. These types of behaviors are called system actions [17].

Some of the common uses:

- Diagnosing process issues
- Hospital applications
- Lab test performance improvement ideas
- Evaluating capital investment decisions
- Network simulators

3.5 Agent based

An ABM is a simulation of an environment where autonomous agents interact with each other and with their surroundings. These agents may be individuals, companies, countries, or any other type of entity. By using a mathematical representation of the

environment, the agent behaviors can be simulated and analyzed. Agents may act according to certain rules, and those rules may change over time. The main goal of an ABM is to simulate the dynamics of a real-world problem.
ABMs can also be called individual-based models or IBMs. Individual-Based Models are different from other types of models because they are based on real life processes. Instead of focusing on abstractions, IBMs focus on the details of the system. Individual-based models are often used to study complex ecological systems like ecosystems, animal populations, or even cities. An IBM simulates individuals interacting with each other and with their environment. These interactions lead to emergent properties of the system.
An agent-based model is a type of micro-scale model that simulates the simultaneous operations and interactions between multiple agents. These agents are often referred to as agents because they act like individual entities within the larger system. Agent-based models are often used to simulate the behavior of real-world systems. For example, agent-based modeling is commonly used to study the spread of diseases, the dynamics of financial markets, and the evolution of social networks.
ABM is an approach to modeling a system where each individual entity within the system is modeled as an agent. Each agent is capable of making decisions about its actions, and the interactions between agents determine the overall behavior of the system. Agents are often assumed to be boundedly rational, meaning they act in ways that maximize their personal benefits. However, many other types of agents exist, including those that are not necessarily self-interested. For example, a swarm of bees might collectively decide to follow a scent trail to a source of nectar, even though the individual bees cannot see the trail.

Most agent-based models are composed of:

- Numerous agents specified at various scales (typically referred to as agent-granularity)
- Decision-making heuristics
- Learning rules or adaptive processes
- An interaction topology
- An environment

Agent based models are usually built as computer simulations and the software is then used to test how changes in individual actions will influence the whole system [14].

4 Implementation

For the development of this project we decided to use the programming language named Java as it is relatively fast and also well maintained furthermore it is the language that was first introduced to me when I began learning the basics of programming and I felt the most comfortable with it. When I found out there is no pullback by selecting Java for development it was a "no-brainer" for me. Java is a high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible.
The IDE which we used to program with was Jet-Brain's IntelliJ which is an integrated development environment written in Java for developing computer software written in Java, Kotlin, Groovy, and other JAR based languages. I used it over the course of my studies and I find it very intuitive and easy to handle.
For version control of the project we used Git which is software for tracking changes in any set of files, usually used for coordinating work among programmers collaboratively developing source code during software development. Its goals include speed, data integrity, and support for distributed, non-linear workflows. Git is a crucial tool for any developer it makes tracking changes and documenting your workflow much easier and more efficient.

4.1 Architecture

The discrete time agent based simulation as selected for the architecture part, which means that every entity of the simulation is it's own actor which handles it's own render and tick function. So they all calculate their own parameters each step and if the GUI is enabled also draw themselves [Figure: 4.1].

The project was divided in into the following parts:

- Actor
- Simulation
- Network
- Intersection

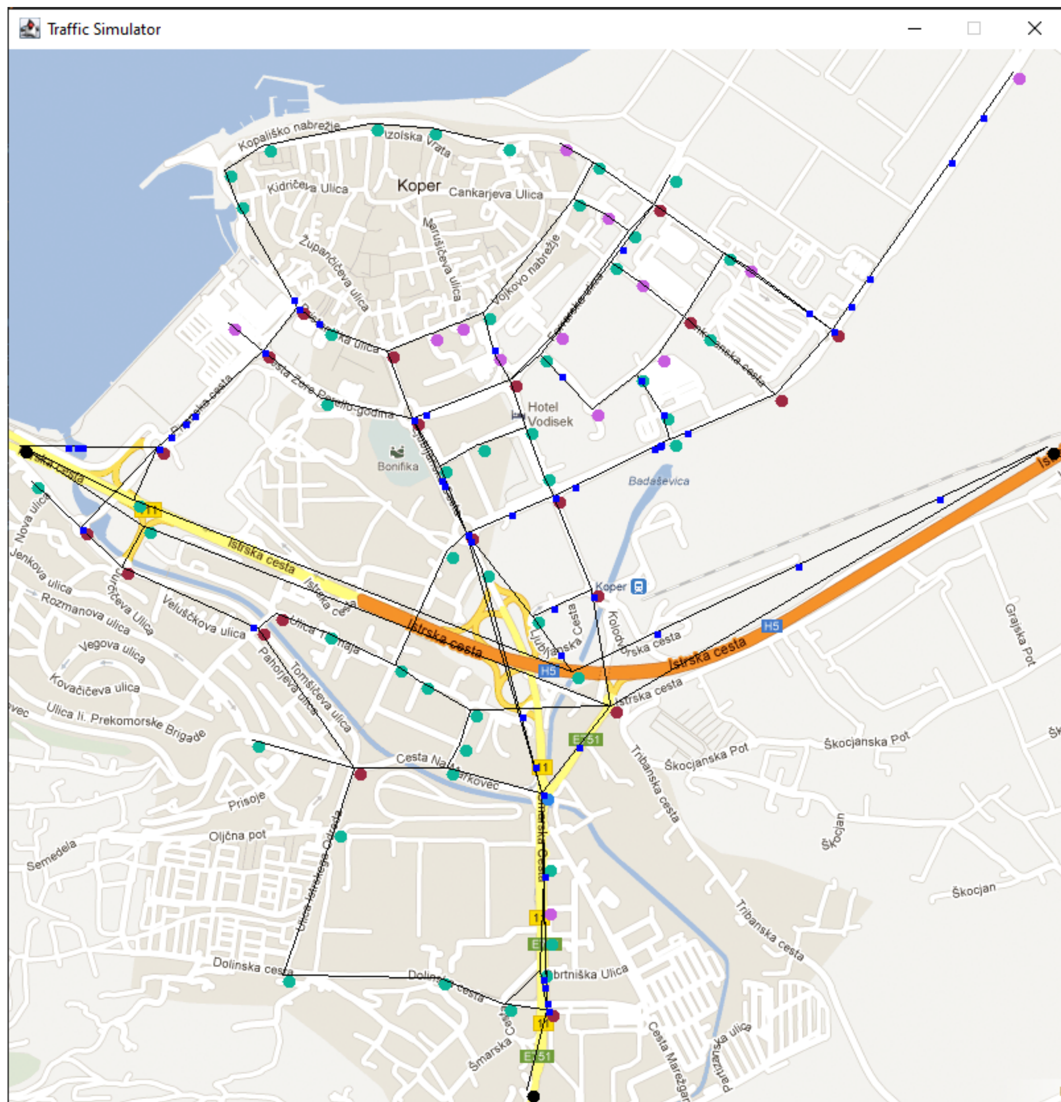

Figure 2: This picture shows the GUI of the simulator which represents the map of Koper and all of the nodes and connections on which the traffic flows.

- Road
- Vehicle

This kind of project's organisation makes it easier to follow through and also makes future adaptations much easier and more straightforward.

4.2 Actor

The actor class is the base class which gets inherited by every agent of the simulation. The class has the base variables and methods which all of the agent classes need to poses and override for there personal performance [Listing: 1].

```
import java.awt.*;

public abstract class Actor {
    float x;
    float y;
    Simulation sim;

    public Actor(float x, float y, Simulation sim) {
        this.x = x;
        this.y = y;
        this.sim = sim;
    }

    public abstract void tick(double elapsedTime) throws
    InterruptedException;
    public abstract void render(Graphics graphics,double elapsedTime);
}
```

Listing 1: The definition of the Actor super class for all the agents of simulation

4.3 Simulation

The simulation class is the connection between all actors so it handles all the information that needs to be exchanged among all of the agents. It also handles the main run method which simulates real world traffic by representing ticks as one real world second. All in all the focus of the simulation class is to handle universal things that are standardized for the whole simulation and keeps up the run method [Listing: 2].

4.4 Vehicle

The Vehicle class is representation of a vehicle that is driving during the simulation. We have also generalized the problem to only include one type of vehicle and also we do not simulate the acceleration and breaking portion for any of the vehicles. Thus the margin of error is the same for every run of the simulation, so we do not jeopardize the necessary comparative analysis of the gathered data. At first we started building the simulator such that every vehicle had constant speed but after doing a couple of tests we concluded that it is more realistic to give every road a speed and then make the vehicles inherit that speed when it arrives to that section of their path.
So every vehicle has a route field which is a queue of roads, the route gets generated

```
public void run(){
        while (running) {
            network.emit(ticks);
            try {
                tick();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            if (this.GUI) render();
            ticks++;
        }
    }
```

Listing 2: The run function of the Simulation class which keeps the simulation going

upon creation of said vehicle in the network generation process, more precisely when the car gets emitted into the simulation. Vehicles have a flag that specifies if the vehicle is driving or waiting (at an intersection), a flag also signals the end of the route thus done with the simulation. These flags are useful for calculating the fitness function of the simulations and are the tools that generate the necessary data for Genetic algorithm application [Listing: 3].

The vehicle goes through the route queue and when it travels from a node on a specific route to the other node the route is then popped off the queue until the queue of roads is empty that means the cars has finished it's route and is done with the simulation. The cars detect when they have arrived at the end of a specific road by calculating if they are geometrically inside the radius of the node that defines the end of the road. Every car also calculates their own steps for each of the ticks so they themselves are responsible for their movement through out the simulation. Once the vehicle arrives at the targeted node it sets it riding flag to false and puts itself on the queue of the targeted node where it waits to get processed by the intersection [Listing: 4].

4.5 Network

In the Network class we read the data about the roads and intersections which is stored in two separate files which are in JSON format. After we gather all of the data the network class generates a directed weighted pseudo-graph with the help of jgraph maven repository [Listing: 5].

```
@Override
public synchronized void tick(double elapsedTime) throws InterruptedException {
    if (this.route.isEmpty() || this.x < 0 || this.y < 0 ||
    this.x > 800 || this.y > 800) {
        this.isRiding = false;
        this.isFinished = true;
        sim.totalTicksWaiting += waiting;
        sim.actors.remove(this);
        if (!route.isEmpty()) System.exit(0);
    }
    if (!this.isRiding || this.isFinished) {
        if (!this.isFinished){
            waiting++;
        }
        return;
    }
```

Listing 3: The section of the Vehicle class which defines the stop conditions

Next core functionality of the Network class is handling the emitters according to distribution data that was collected with the help of a website called *promet.si* that uses counters for the main entrances into Koper. By monitoring the counters for the three main points of entry into the city every three hours we got the generalized distribution of the cars that are entering the city on average every day. So we collected the base flow of the network.

4.5.1 Dijkstra algorithm

When the network emits cars it also generates an instance of the Vehicle class and assigns a path from the emitter to the destination node somewhere in the city. For the calculation of the path we use Dijkstra algorithm which will be shortly described below.

Dijkstra's algorithm finds the shortest path between two nodes in a graph. This algorithm is used to solve problems such as finding the fastest route between cities or determining the quickest time to get from A to B. It was invented in 1956 by computer scientist Edsger W. Dijkstra [13].
For a given supply node in the graph, the algorithm finds the shortest direction between that node and each other. It can additionally be used for discovering the shortest paths from a single node to an specific node via stopping the algorithm as soon as the shortest route to the specified node has been determined. For example, if the nodes of the design characterize cities and side route expenses signify using distances between pairs of cities linked through a direct street, Dijkstra's algorithm can be used to locate the shortest

```
Road currentRoad = this.route.peek();
this.speed = currentRoad.getSpeed();

float totalTicks = currentRoad.getLength() / this.speed;

float oneStepX = ((currentRoad.getStart().x - currentRoad.getEnd().x)
/ totalTicks)*(-1);
float oneStepY = ((currentRoad.getStart().y - currentRoad.getEnd().y)
/ totalTicks)*(-1);

float oneRealStepX = ((float) (oneStepX * elapsedTime));
float oneRealStepY = ((float) (oneStepY * elapsedTime));


//--------------CIRCLE AROUND INTERSECTIONS CALCULATION----------------
float firstParameter = (currentRoad.getEnd().x - this.x)*
(currentRoad.getEnd().x - this.x);
float firstParameterNext = (currentRoad.getEnd().x - this.x+oneRealStepX)*
(currentRoad.getEnd().x - this.x+oneRealStepX);
float drugiParameter = (currentRoad.getEnd().y - this.y)*
(currentRoad.getEnd().y - this.y);
float drugiParameterNext = (currentRoad.getEnd().y - this.y+oneRealStepY)*
(currentRoad.getEnd().y - this.y+oneRealStepY);

float sum = firstParameter + drugiParameter;
float sumNext = firstParameterNext + drugiParameterNext;


float d = 10*10 - sum;
float dNext = 10*10 - sumNext;
//--------------------------------------------------------------------

//If radius squared of the intersection is >= 0 then we are at the intersection...
if (d > 0 || this.next) {
    this.comingFromArc = currentRoad.getEndArc();
    this.nextRoad();
    this.sim.getIntersection(currentRoad.getEndId()).
    arrived(this.comingFromArc, this);
} else {
    if (dNext > 0) this.next=true;
    this.x = this.x + oneRealStepX;
    this.y = this.y + oneRealStepY;
}
}
```

Listing 4: The section of the Vehicle class which calculates steps for every tick of the simulation

route for any problem that can be symbolised with vertexes and connections [16].

4.5.2 Emit functionality

The core of the Network class is the emit method which handles the flow of the network and how often the vehicles get generated during the time of the simulation. We gather the intervals on which vehicles enter the simulation through the distribution file which contains the average flow of the traffic in the city of Koper and it's close surroundings. The distribution data is necessary because it allows us to generate the traffic of the city close to real conditions, so we do not just generate random number of cars at random times or even worse generate all the planned cars at the beginning at once.
By using the emitters we get close to real conditions of the traffic flow in Koper, but we cannot just generate the flow of the vehicles into the city, that is why we calculated the flow out of the city too. When we gathered all the data to make a solid distribution we also assumed that about 30% of the traffic happens inside of the city by daily commuters and people going to work. By using this assumption in our emit function we got: 35% of the vehicles with a path from and inner city node to and emitter which means it is exiting the city, about 35% of the cars are entering the city by having a path from an emitter node to the inner city node and the 30% left of the traffic flow is as mentioned before inner city flow that goes form one inner city node to another [Listing: 6].

4.6 Intersections

The most important part of the simulator is the Intersection class which is implemented in a way that it is easily interchangeable thus the genetic algorithm can be applied to the simulator with ease. As it was mentioned before, the vehicles upon arrival to an intersection are put in a queue. Every node has a list of queues which represent the entry points of the intersections. The vehicles select the queue corresponding the intersection's entry edge/road. Three types of intersections are implemented: regular intersection, the semaphore intersection and the roundabout intersection, each type will be further explained below.

4.6.1 Regular intersection

The regular intersection it is just an implementation of an intersection where there is only right rule applied and there are priority roads. That means that if a car is alone at the intersection the node informs the vehicle it can proceed with it's route. There is a case when the car is on a priority road that means it can always proceed as well,

```
listType = new TypeToken<ArrayList<RoadData>>() {}.getType();
reader = new JsonReader(new FileReader(this.roads_file));
List<RoadData> roadsData = new Gson().fromJson(reader, listType);
List<Road> roads = new ArrayList<Road>();
roadsData.forEach(R-> roads.add(new Road(R,simulation)));
roadMap = new HashMap<>(roads.size());
roads.forEach(road -> {
    roadMap.put(road.getId(), road);
});
graph = new DirectedWeightedPseudograph<>(Road.class);

//add intersections as vertexes
intersections.forEach(intersection -> graph.addVertex(intersection));
//add edges by mapping them to vertices
roads.forEach(road -> {
    if(road != null) {
        graph.addEdge(

        intersectionMap.get(road.getStartId()), //start
        intersectionMap.get(road.getEndId()), //end
        road
        );
        graph.setEdgeWeight(road, road.getLength());

        //Adds incoming and outgoing roads of an intersections
        intersectionMap.get(road.getEndId()).addIn(road);
        intersectionMap.get(road.getStartId()).addOut(road);

        //initializing intersection queues for every incoming road
        intersectionMap.values().forEach(Intersection::initialize);
    }
});

parking = intersectionMap.values().stream().
filter(intersection -> intersection.getType() == 0).
collect(Collectors.toCollection(LinkedList::new));
dijkstraShortestPath = new DijkstraShortestPath(graph);
```

Listing 5: This code describes the generation of a directed weighted pseudo-graph and all the connections between the nodes

```
public void emit(long ticks) {
Random rnd = new Random(config.seed);
if (ticks >= config.timeInSec) {//86400 seconds in a day
    System.out.println(simulation.totalTicksWaiting/ simulation.totalCars);
    System.exit(0);
}
if (ticks % 10800 == 0) k++;
Vector<Vehicle> cars = new Vector<>();

for (Emitter emitter: distribution.emitters) {
    if (emitter.spaceDrivingIn[k%8] != 0 &&
    ticks % emitter.spaceDrivingIn[k%8] == 0) {
        CarsIn(rnd, cars, emitter);
    } else if (emitter.spaceOvertakingIn[k%8] != 0 && ticks %
    emitter.spaceOvertakingIn[k%8] == 0) {
        CarsIn(rnd, cars, emitter);
    } else if (emitter.spaceDrivingOut[k%8] != 0 && ticks %
    emitter.spaceDrivingOut[k%8] == 0) {
        CarsOut(rnd, cars, emitter);
    } else if (emitter.spaceOvertakingOut[k%8] != 0 && ticks %
    emitter.spaceOvertakingOut[k%8] == 0) {
        CarsOut(rnd, cars, emitter);
    }
}
for (int i = 0; i < Math.round((cars.size()/0.7)*0.3); i++) {
    Collections.shuffle(parking, rnd);
    List<Road> route;
    while ((route = dijkstraShortestPath
    .getPath(parking.getFirst(), parking.getLast())
    .getEdgeList()).isEmpty()) {
        Collections.shuffle(parking, rnd);
    }
    Vehicle car = new Vehicle(14, route, simulation);
    cars.add(car);
}
simulation.totalCars += cars.size();
simulation.actors.addAll(cars);
}
```

Listing 6: This code describes the networks emit function and it’s car generation

the same goes when there are two cars at the intersection and they go in opposite directions of each other. The case where there are two cars and neither of them is on the priority road then the right-hand rule is applied to decide which car is to proceed in this step of the simulation and which goes in one of the proceeding ones. We also limited the intersections to 4 entries to simplify the process of the intensity of the calculations, that means that the node can have at most have four incoming and four outgoing roads [Listing: 7].

4.6.2 Semaphore

The implementation of the semaphore intersection is straightforward, there is a parameter which is semaphore timer that delegates when the lights switch on different lanes and we have a Boolean flag that can be true or false, which dictates if odd arc roads have green light or even arc roads have green light. We also do not implement the yellow light as it does not change the outcome of the simulation at all, which means after the timer resets if the odd arcs had a green light it switches their light to red and the even arcs lights to green [Listing: 8].

4.6.3 Roundabout

The last intersection type implemented is the roundabout which is implemented with a simple array that represents slots inside a roundabout. So we have an array that has two empty slots between every incoming road of the intersection. If the slot before the road from which the car is coming is empty and also the slot which the vehicle wants to occupy then the car can enter the roundabout. The cars in the roundabout move by one slot each step of the simulation and on the step they come to the road that is next in their route they exit the roundabout. This is a simplified implementation of a roundabout but it is effective and easily scale-able if we ever want to have intersections with more then four entry roads or switch to larger cities [Listing: 9].

4.7 Genetic Algorithm

A genetic algorithm (GA) is an optimization technique based on Darwinian principles. It mimics the process of natural selection through trial and error. In order to find the optimal solution, a population of candidate solutions is evaluated according to some fitness function. Those candidates that perform best survive to reproduce and produce offspring. These new offspring replace those that performed poorly. This cycle continues until no further improvement can be made [Figure: 4.7].

```
switch (type) {
    //Basic intersection
    case 1:
        if (onTheIntersection.size() == 1) {
            vehicleQueue.get(onTheIntersection.get(0).getComingFromArc()).remove();
            onTheIntersection.get(0).setRiding(true);
            return;
        }
        if (!vehicleQueue.get(this.arc1).isEmpty() ) {
            Vehicle arc1Car = vehicleQueue.get(this.arc1).peek();
            if (arc1Car != null) {
                vehicleQueue.get(arc1Car.getComingFromArc()).remove();
                arc1Car.setRiding(true);
            }
            return;
        }
        //<< NO CAR ON PRIORITY ROAD AND MORE THAN ONE CAR ON THE INTERSECTION>>
        int endArc = onTheIntersection.get(0).getRoute().peek().getStartArc();
        if (onTheIntersection.stream()
        .allMatch(x -> x.getRoute().peek().getStartArc() == endArc)) {
            onTheIntersection.forEach(x -> {
                if (x.getComingFromArc() == 0 && endArc == 3) {
                    vehicleQueue.get(x.getComingFromArc()).remove();
                    x.setRiding(true);
                } else if (endArc == x.getComingFromArc() - 1) {
                    vehicleQueue.get(x.getComingFromArc()).remove();
                    x.setRiding(true);
                }
            });
            return;
        }
        // If we come to here all the cars on the intersection can go
        onTheIntersection.forEach(v -> {
            vehicleQueue.get(v.getComingFromArc()).remove();
            v.setRiding(true);
        });
        break;
```

Listing 7: This is the code of the regular intersection type and it’s rules

```
//Semaphore
case 2:
    if (semaphore) {
        onTheIntersection.forEach(x -> {
            if (x != null) {
                if (x.getComingFromArc() % 2 == 0) {
                    vehicleQueue.get(x.getComingFromArc()).remove();
                    x.setRiding(true);
                }
            }
        });
    } else {
        onTheIntersection.forEach(x -> {
            if (x != null) {
                if (x.getComingFromArc() % 2 == 1) {
                    vehicleQueue.get(x.getComingFromArc()).remove();
                    x.setRiding(true);
                }
            }
        });
    }
    semaphoreTimer--;
    if (semaphoreTimer == 0) {
        semaphoreTimer = 40;
        semaphore = !semaphore;
    }
    break;
```

Listing 8: This is the code of the semaphore intersection type and it’s rules

```
//Roundabout
case 3:
    if (!(Arrays.stream(roundabout).allMatch(Objects::isNull))) shift();
    for (int i = 0; i < roundabout.length; i++) {
        if (roundabout[i] == null) continue;
        Vehicle vehicle = roundabout[i];
        if (i % 3 == 0) {
            if (i / 3 == vehicle.getRoute().peek().getStartArc()) {
                roundabout[i] = null;
                vehicle.setRiding(true);
            }
        }
    }
    onTheIntersection.forEach(x -> {
        int arc = x.getComingFromArc();
        int entrance = arc * 3;
        int going = arc * 3 - 1;
        int coming = arc * 3 + 1;
        if (arc * 3 == 0) {
            going = roundabout.length - 1;
        }
        if (roundabout[coming] == null &&
        roundabout[going] == null && roundabout[entrance] == null) {
            vehicleQueue.get(arc).remove();
            roundabout[going] = x;
        }
    });
    break;
```

Listing 9: This is the code of the roundabout intersection type and it’s rules

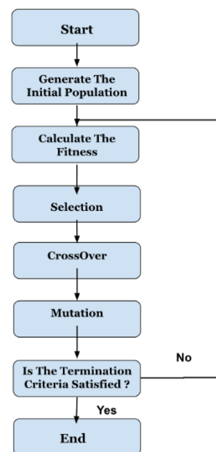

Figure 3: This diagram shortly describes a flow of a basic genetic algorithm so it can be visualized

These structures, called Individuals, can represent solutions to Problems, Strategies for Playing Games, Visual Images, or Computer Programs. Genetic Algorithms are loosely based on ideas from Population Genetics. First, a population of individuals is created randomly. In the simplest case, each individual is a bit string and can be thought of as a candidate solution for some problem of interest.
Variations among individuals within a population can lead to some individuals being superior to others. For example, if an individual is better at solving problems than another one, then they may be selected for reproduction. In order to select individuals for reproduction, we create a new population by copying those who are best at solving problems and removing those who are worst. However, the copies aren't exact. There is a chance of mutation (random bit flipping) or crossover (exchanging parts of the string).
By transforming the previous sets of good individuals to a newer one, the mutation and cross-over operations generate a new set or samples that ideally have a better chance of also being good than the original ones. When this cycle of evaluating, selecting, and genetic operations is repeated for many generations, the final outcome is an improved set of individuals that represent improved solutions to whatever problem was posed [5].

In our case the GA algorithm is applied on top of the simulator built and the fitness function is based on the optimization of the traffic flow in the specified network we

are trying to optimise. We define an individual as one instance of the simulator and genomes are the intersections of the simulation which can be modified and mutated. After every simulation is done we get an average wait time of vehicles in that specific instance and then we take the best ones and cross them over and mutate them. This process continues until a certain threshold is met then the GA stops and we get a solution.

4.8 Results

The first run of the simulator with the genetic algorithm produced this results, which are demonstrated with the graph and data below [Figure: 4.8].

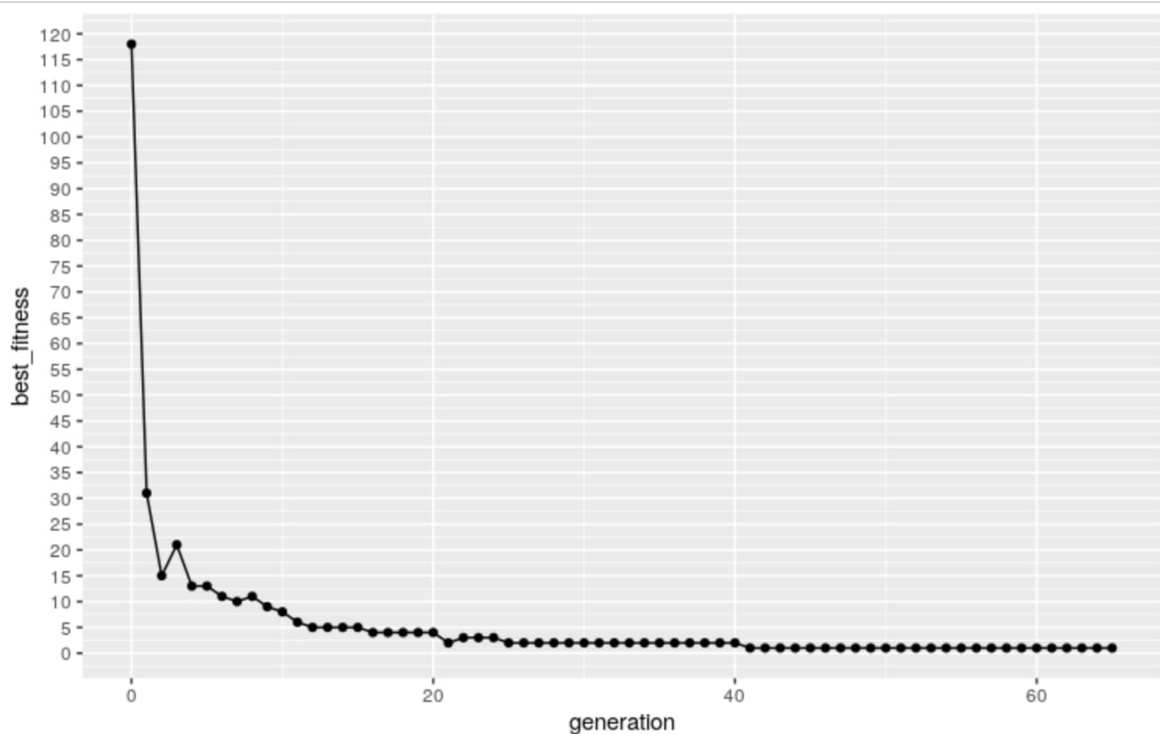

Figure 4: The first run of the Simulator with GA applied

The intersection distribution was as follows:

- 13 are type 0
- 45 are type 1
- 17 are type 2
- 4 are type 3

The optimization of the network works very well on the first generation which can be just a coincidence that we chose a good seeded random and everything aligned, but all in all the result show that the the traffic has improved over time with different crossovers and mutations.

5 Conclusion

The optimization of urban traffic will over the next year be an ever growing problem as the population on the Earth grows and there are more and more vehicles being used and condensed into not so flowing urban networks. That is why we will need more and more elaborate ways of constructing urban traffic networks which can handle a large inflow and outflow of vehicles at one time. Which means we will have to build highly modifiable and detailed simulators which will be able to very precisely predict and analyse a large network and also alongside that produce improved and more sophisticated genetic algorithms. For the future the goal is to expend the functionality of both the simulator and genetic algorithms in a way that we can provide different parameters which we want to base the calculation on, such as optimization based on flow, emissions, money and so on. Also we want to have different genetic algorithms to apply on top of the simulator which will allow us to compare results and come to a generally more effective solution.

6 Bibliography

[1] Jaime Barceló and Jordi Casas. Dynamic network simulation with aimsun. In *Simulation approaches in transportation analysis*, pages 57–98. Springer, 2005. *(Cited on page 3.)*

[2] Jaume Barceló. *Fundamentals of Traffic Simulation*. Springer New York, NY, 2010. *(Cited on page 5.)*

[3] Michael Behrisch, Laura Bieker, Jakob Erdmann, and Daniel Krajzewicz. Sumo–simulation of urban mobility: an overview. In *Proceedings of SIMUL 2011, The Third International Conference on Advances in System Simulation*. ThinkMind, 2011. *(Cited on pages V in 2.)*

[4] Moshe Ben-Akiva, Haris N Koutsopoulos, Tomer Toledo, Qi Yang, Charisma F Choudhury, Constantinos Antoniou, and Ramachandran Balakrishna. Traffic simulation with mitsimlab. In *Fundamentals of traffic simulation*, pages 233–268. Springer, 2010. *(Cited on page 3.)*

[5] Stephanie Forrest. Genetic algorithms. *ACM Computing Surveys (CSUR)*, 28(1):77–80, 1996. *(Cited on page 27.)*

[6] R Grau and J Darcelo. Getram: A generic environment for traffic analysis and modeling. *IFAC Proceedings Volumes*, 27(12):701–706, 1994. *(Cited on page 3.)*

[7] David C Joy. An introduction to monte carlo simulations. *Scanning microscopy*, 5(2):4, 1991. *(Cited on pages 5 in 6.)*

[8] Sven Maerivoet and Bart De Moor. Cellular automata models of road traffic. *Physics reports*, 419(1):1–64, 2005. *(Cited on page 6.)*

[9] Johannes Nguyen, Simon T. Powers, Neil Urquhart, Thomas Farrenkopf, and Michael Guckert. An overview of agent-based traffic simulators. *Transportation Research Interdisciplinary Perspectives*, 12:100486, 2021. *(Cited on pages 8 in 9.)*

[10] Jean-Paul Rodrigue. *The geography of transport systems*. Routledge, 2020. *(Cited on page 1.)*

[11] Eduardo Sáez and Eduardo González-Olivares. Dynamics of a predator-prey model. *SIAM Journal on Applied Mathematics*, 59(5):1867–1878, 1999. *(Cited on page 10.)*

[12] Saidallah, Mustapha, El Fergougui, Abdeslam, and Elalaoui, Abdelbaki Elbelrhiti. A comparative study of urban road traffic simulators. *MATEC Web Conf.*, 81:05002, 2016. *(Cited on pages V, 2, 3 in 4.)*

[13] Moshe Sniedovich. Dijkstra's algorithm revisited: the dynamic programming connexion. *Control and cybernetics*, 35(3):599–620, 2006. *(Cited on page 18.)*

[14] Wikipedia. Agent-based model, 2022. [Online; accessed 25-July-2022]. *(Cited on page 13.)*

[15] Wikipedia. Continuous simulation, 2022. [Online; accessed 19-July-2022]. *(Cited on pages 10 in 11.)*

[16] Wikipedia. Dijkstra's algorithm, 2022. [Online; accessed 26-July-2022]. *(Cited on page 20.)*

[17] Wikipedia. Discrete-event simulation, 2022. [Online; accessed 19-July-2022]. *(Cited on page 12.)*

[18] Wikipedia. Simulation software, 2022. [Online; accessed 19-July-2022]. *(Cited on page 11.)*

[19] Wikipedia. Traffic simulation, 2022. [Online; accessed 08-August-2022]. *(Cited on page 5.)*