2022

ZAKLJUČNA NALOGA (FINAL PROJECT PAPER)

DELJANIN

UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN
INFORMACIJSKE TEHNOLOGIJE

ZAKLJUČNA NALOGA
(FINAL PROJECT PAPER)

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

PETAR DELJANIN

UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN INFORMACIJSKE TEHNOLOGIJE

Zaključna naloga
(Final project paper)

Razvoj genskega algoritma za optimizacijo topologije prometnega omrežja z uporabo simulatorja na podlagi agentov (Developing a genetic algorithm for optimizing traffic network topologies by using an agent-based simulator)

Ime in priimek: Petar Deljanin
Študijski program: Računalništvo in informatika
Mentor: izr. prof. dr. Jernej Vičič
Somentor: assist. Aleksandar Tošić

Koper, september 2022

Ključna dokumentacijska informacija

Ime in PRIIMEK: Petar DELJANIN

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

Kraj: Koper

Leto: 2022

Število listov: 35 Število slik: 9 Število referenc: 38

Mentor: izr. prof. dr. Jernej Vičič

Somentor: assist. Aleksandar Tošić

Ključne besede: genetski algoritmi, cestno omrežje, optimizacija, simulator.

Izvleček:
Obstaja globalna težava v zvezi z naraščajočim številom vozil in stanjem obstoječe infrastrukture cestnega omrežja [36]. Vedno povečavajoče število osebnih in dostavnih vozil predstavlja vedno povečavajočo težavo za infrastrukturo, kar povzroča vse pogostejše prometne zastoje [2]. Cestna omrežja so ključna za gospodarsko rast vsake države [30]. Nujna je strateška širitev, ustrezno vzdrževanje in optimizacija teh omrežij, da se zagotovi učinkovita povezava znotraj geografske regije.
Raziskali bomo možne rešitve tega problema. S spremembo tipov križišč bomo spreminjali topologijo cestnega omrežja, da bi povečali pretočnost. Rešitev problema je težko najti, saj bi iskanje najboljšega cestnega omrežja za določeno regijo trajalo zelo dolgo. Zaradi narave problema in ker je računsko neizvedljivo razviti neposredno rešitev je iskanje optimalne rešitve neizvedljivo. V takih primerih je nujen hevristični pristop. Torej, bomo raziskovali uporabo genetskih algoritmov [26] za učinkovito iskanje prostora možnih rešitev. Kot ciljna funkcija bo uporabljen prometni simulator, prilagojen problemu (zagotavlja simuliran pretok za rešitev).
Naš pristop bomo potrdili s poskusom optimizacije obstoječega cestnega omrežja mesta Koper [37], Slovenija.

Key words documentation

Name and SURNAME: Petar DELJANIN

Title of final project paper: Developing a genetic algorithm for optimizing traffic network topologies by using an agent-based simulator.

Place: Koper

Year: 2022

Number of pages: 35 Number of figures: 9

Number of references: 38

Mentor: Assoc. Prof. Jernej Vičič, PhD

Co-Mentor: Assist. Aleksandar Tošić

Keywords: genetic algorithms, road network, optimization, simulator.

Abstract:
There is a global problem regarding the increasing number of vehicles and the state of existing road network infrastructure [36]. The ever-growing number of personal and delivery vehicles presents an ever-growing problem to the infrastructure, resulting in more frequent traffic congestion [2]. Road networks are essential for the economic growth of every country [30]. It is necessary to make strategic expansion, adequate maintenance, and optimization of these networks to guarantee an efficient connection within a geographic region.
We will be exploring potential solutions to this problem. By changing intersection types we will be changing the road-network topology in order to increase the throughput. The solution to the problem is difficult to discover, since finding the best road-network for a given region would take a very long time.
The nature of the problem makes searching for the optimal solution infeasible, because it is computationally infeasible to develop a direct solution for. In such cases, a heuristic approach is necessary. Thus, we will explore the use of Genetic algorithms [26] to efficiently search the space of possible solutions. As an objective function, a traffic simulator tailored to the problem will be used (delivering a simulated throughput for the solution).
We will validate our approach by attempting to optimize the existing road network of the town of Koper [37], Slovenia.

Acknowledgements

I would like to thank the mentor doc. dr. Jernej Vičič and the co-mentor mag. Aleksandar Tošić for the opportunity to work with them on this amazing project, and for their help and mentorship in my studies at the faculty.

I would like to express my deepest gratitude to my co-mentor mag. Aleksandar Tošić for being not just a faculty mentor, but for being a personal development mentor to me as well as has helped me immensely through-out my personal and professional struggles in life, in the past few years.

I am also grateful to mag. Jordan Aiko Deja for guiding me and helping me find good job opportunities at the beginning of my career.

I would like to extend my sincere thanks to my colleagues at the faculty. Especially to my great friends Jurij Požrl and Patrik Matošević for co-creating such a wonderful experience during my studies.

Lastly, I would be remiss in not mentioning my family, especially my parents. Their belief in me has kept my spirits and motivation high during this process, and I wouldn't have gotten this far with my studies without their support.

Contents

Table of Figures

1 Introduction

The work of Church, Gödel, and Turing was one of the more significant achievements of twentieth-century mathematics, particularly in the fields of logic and computer science. In the 1930s, they provided a precise definition of what it means for a problem to be computationally solvable. Which showed that there are incomputable problems in logic and computer science [35]. Because there is no reliable, exact method to solve certain problems and because some problems are so complex that they are computationally infeasible to develop a solution for, we separated them into complexity classes. These classes help scientists in grouping the problems based on how much time and space they require to solve them and verify the solutions. Therefore, we have these complexity classes:

- P class

  If a deterministic Turing machine can solve a problem in polynomial time, that problem belongs in the complexity class P. Every problem in this class has a solution which takes polynomial time on the input size n. I.e. $f(n)$ is of form $x_k n_k + x_k - 1 n_k - 1 + ... + x_2 n^2 + x_1 n + x0$ such that $x_k, ..., x_0$ are constant factors (possibly even 0). The polynoms order is the largest exponent k such that $x_k \neq 0$ [14].

- E Class

  If a deterministic Turing machine can solve a problem in exponential time, that problems belongs in the complexity class E. An example of the time complexity would be: $f(n) = 3^n$ or $f(n) = n^2 n$. It does not matter, if $f(n)$ contains a polynomial part, because the exponential part (n is in the exponent) is largely dominating the complexity. An example would be: $f(n) = 2^{n/6} + 7x^9$ has the complexity $O(2^{n/6}) = O(2^n)$, which is an exponential function. Furthermore, if a problem belongs to the complexity class P , then it also belongs to the class E, because $P \subset E$. Note that E isn't the most complex class of problems. An example of a more complex problem would be the problem with the complexity, $O(n!)$ $(n! = n(n-1)(n-2)...1)$ which is more difficult [14].

- NP Class

  The problems that can be solved by a non-deterministic Turing machine in polynomial time belong to the NP complexity class. The word "Nondeterministic" means there may be several ways for the computation to be executed. [35] Class NP exists between the class P and the class $E : E \subseteq NP \subseteq E$. Meaning that all problems in the complexity class NP also belong to the class E, and a deterministic Turing machine (or a computer program) can solve it in exponential time. It could happen that some of these problems can be solved faster, maybe in polynomial time, and that would mean they actually belong to the class P. But, it is not known if all of them are solvable in polynomial time by some algorithms [14].

- NP-complete Class

  NP-complete problems are a subset of the class NP. A problem x is in the NP-complete class if:

  1. $x \in NP$ (A non-deterministic Turing machine problem can solve the problem in polynomial time)
  2. All other problems in class NP can be reduced to the problem x in polynomial time.

  This means that in the complexity class NP the NP-complete problems are the most difficult problems. If just one of them is solvable in polynomial time, this would imply that we could solve all the problems in the class NP in polynomial time [14].

- NP-hard Class

  The NP-hard don't belong in class NP, but all problems in class NP can be reduced to them. We can see that NP-complete class problems are a subset of the NP-hard class, and that's why NP-complete class problems are sometimes called NP-hard as well [14].

  To summarize:
  $P \subseteq NP \subseteq E$
  NP-complete $\subseteq NP$
  NP-complete $\subsetneq NP$-hard

Now we will look at a subset of problems. These are the optimization problems. Any arbitrarily chosen optimization problem can be defined as objective and feasibility functions and as an instance drawn from a finite set A. The goal is to try to find a value in the set A such that we obtain the maximum (or minimum) value of the objective function subject. The formal definition of the optimization problems can be found in [18] and [28].

Heuristic optimization is mostly used to solve this kind of problems. The word "heuristic" derives from Greek "herusken". It means "to discover." So a heuristic attempts to use the methods and rules of discovery or assisting in problem-solving. Problem-solving is the act of defining a problem; determining the cause of the problem; identifying, prioritizing, and selecting alternatives for a solution; and implementing a solution. Furthermore, heuristics are relatively simple procedures that are designed to find good but not necessarily the best, most optimal solution to the given problem [38]. There are many types of heuristic algorithms in the literature. A short summary of such algorithms follows: alpha-beta search [23], genetic algorithms [27], backtracking, hill-climbing [19], simulated annealing [1], tabu search [12], etc.
Notice the difference between metaheuristic and heuristic algorithms is that heuristics are often problem-dependent, that is, you define a heuristic for a given problem. Metaheuristics are problem-independent techniques that can be applied to a broad range of problems. A heuristic is, for example, choosing a random element for pivoting in the Quicksort (a sorting algorithm). A metaheuristic knows nothing about the problem it will be applied to, it can treat functions as black boxes. The authors of the article [4] describe the black box as: "A black box is a fiction representing a set of concrete systems into which stimuli S impinge and out of which reactions R emerge." Find out more about them in [4].
Metaheuristic algorithms are divided into two types: single solution and population-based metaheuristic algorithms, as depicted in the figure 1 below.

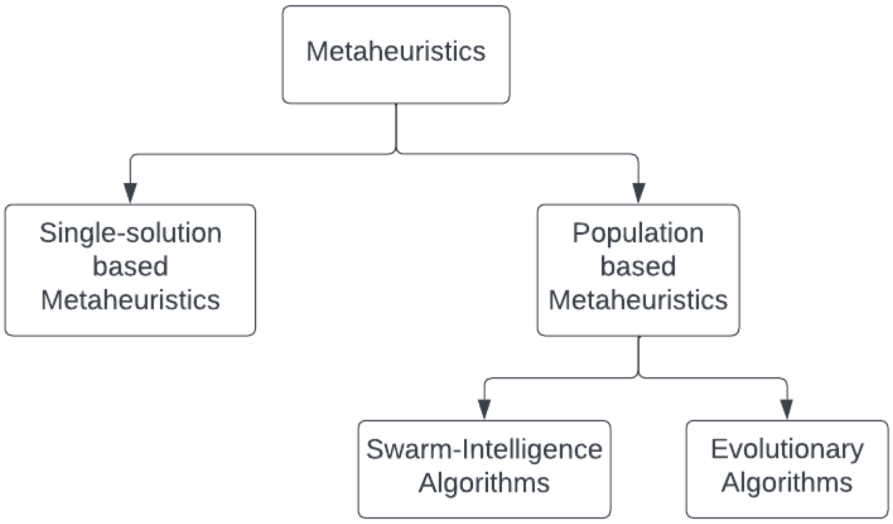
Metaheuristics
Single-solution based Metaheuristics
Population based Metaheuristics
Swarm-Intelligence Algorithms
Evolutionary Algorithms

Figure 1: Classification of metaheuristic Algorithms

Single-solution metaheuristic algorithms use a single candidate solution and improve it using local search. The solution obtained from single-solution based metaheuristics, on the other hand, may become stuck in local optima [21]. Tabu search (TS), microcanonical annealing (MA), simulated annealing and guided local search are examples of some well-known single-solution based metaheuristics. Population-based metaheuristics utilizes numerous candidate solutions during the search process. These metaheuristics preserve population variety and keep solutions from becoming trapped in local optima. Here are some well-known population-based metaheuristic algorithms: genetic algorithm (GA) [25], ant colony optimization (ACO) [8], particle swarm optimization (PSO) [17], spotted hyena optimizer (SHO) [5], seagull optimization (SOA) [7], and emperor penguin optimizer (EPO) [6, 16].
In the following chapters we will explain how genetic algorithms function, what problem we are trying to find a more optimal solution for, and how we developed the optimization software.

2 Genetic Algorithms

2.1 Definition

Genetic algorithms are a type of heuristic optimization or problem-solving technique. They adhere to Darwin's principle of evolution via genetic selection. It is essential to establish the biological background of GAs at this point. These biological terms are used in the spirit of analogy with real biology, however the entities they correspond to here are far simpler.
All living organisms consist of cells, and each cell contains a set of one or more *chromosomes*. These *chromosomes* are strings of DNA that function as a "blueprint" for the organism. A chromosome can be divided into *genes*. They are functional blocks of the DNA, and each encodes a particular protein. From an abstract point of view, we can think of *genes* as encoding a trait, such as eye color. *Alleles* are the distinct possible "settings" of a trait (e.g., blue, brown, green). Each gene is located on the chromosome at a specific *locus* (position). In many organisms, each cell contains multiple chromosomes. The complete collection of these chromosomes (of the genetic material), is called the *genome* of the organism [27].

GAs use a highly abstracted version of the evolutionary process to evolve solutions of a given problem. All GAs work with a population of artificial *chromosomes*. These are strings of a defined finite alphabet (like binary, or octal). In *chromosomes*, each *locus* (specific position on chromosome) has two or more, depending on the alphabet, possible *alleles* (variant forms of genes) - 0 and 1 (in case of binary). *Chromosomes* are regarded as points in the solution space and each of them has a *fitness*, a real number, which represents the quality of the solution, for that particular problem [16, 24].
To simplify, we have the population, which consists of chromosomes, these chromosomes are encoded in a certain way using an alphabet, and they have a fitness value which represent their quality.

2.2 How do genetic algorithms work?

For easier understanding, we'll say individuals instead of chromosomes. Genetic algorithms initiate with a population of individuals chosen at random. Check figure 2 for reference. These are processed using biological-inspired genetic operators, namely selection, crossover and mutation, by iterative replacing of its population. Firstly, GAs evaluate the fitness of each individual in the population. Then they select better individuals (chromosomes) based on their fitness value for further processing. Next, GAs create the offspring by combining the individuals genomes of 2 or more previously selected individuals and then finally, they mutate the genomes, to keep diversity in the population. Then we repeat this sequence of selecting, crossing over and mutating, until some stop criterion is reached, or we become satisfied with the more optimal solution the GA has created.

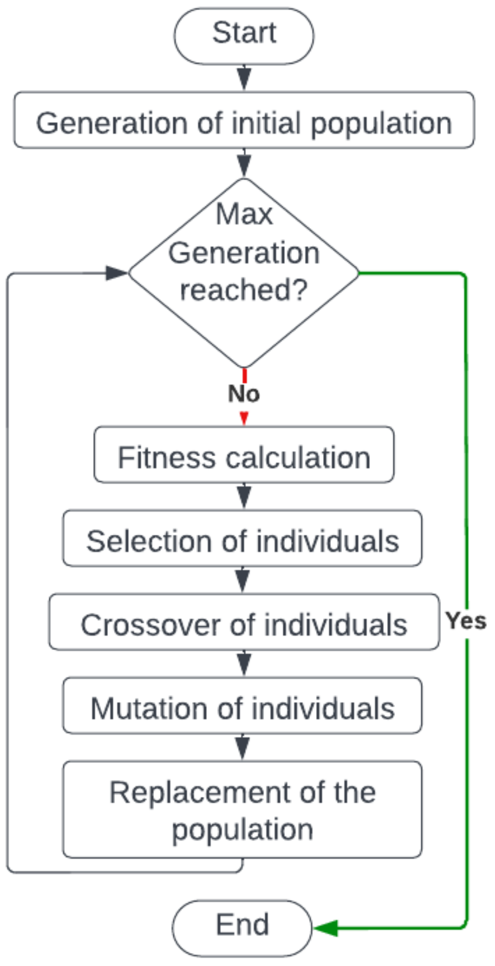

Figure 2: Genetic Algorithm

The string interpretation of Individuals is fully dependent on the problem. A string of 20 bits may represent a single integer value in one problem, yet it may represent the appearance of 20 separate components in some other complicated process. The ability of common representations to be utilized in this manner for a wide range of issues is a

strength of GAs.
Yet, chromosomal encoding will only carry a limited amount of problem-specific information. The fitness function encodes a large portion of the significance of a single chromosome for a certain organism [24].
Before we continue with an even deeper explanation of the specific parts of genetic algorithms, it is valuable to mention some operations used in GAs as they are highly modular in nature, it is important to have an idea of what kind of GAs exist. We can see some of the different "modules" of genetic algorithms in the figure 3 below.

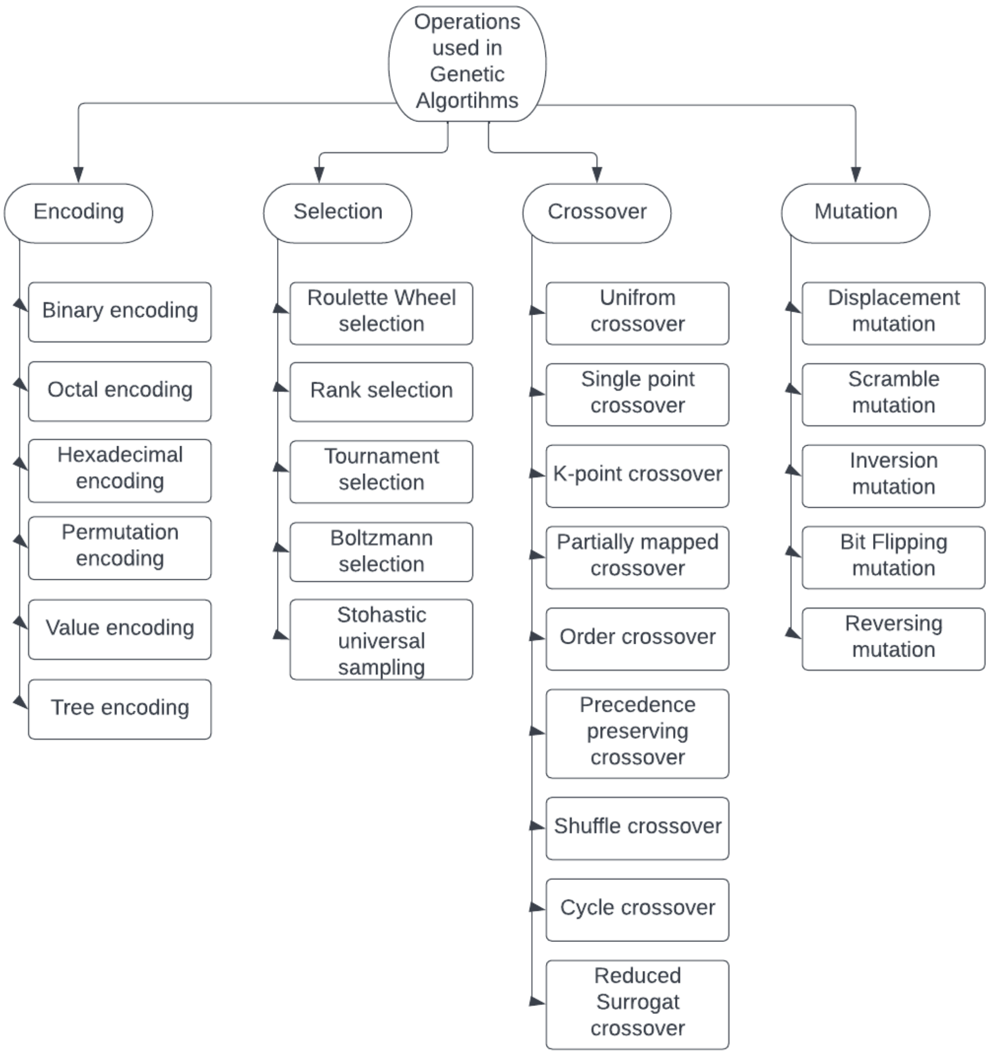

Figure 3: Operations in GA

2.2.1 Chromosome encoding

The encoding techniques are important for most computing problems because we are compiling the high level, human understandable, information into something a computer can interpret. The provided information about the problem has to be encoded into strings of data [13, 22]. Some forms the data can take are binary, octal, hexadecimal, value-based and tree [16].

In **binary encoding**, each individual is defined as a string of 1s and 0s [33]. Each bit in the encoding conveys the individuals' characteristics. It allows for more rapid implementation of crossover and mutation operators. But, converting to binary costs additional effort, and algorithm accuracy is dependent on binary conversion. The bit string is modified according to the problem we are trying to solve. Because of natural representation and epistasis (the interaction of genes that are not alleles), the binary encoding technique is inappropriate for some engineering design challenges [16].
In the **octal encoding** scheme, the individual is represented by strings that consist of octal numbers (0-7) [16]. In the **hexadecimal encoding** technique, the individual is represented by strings of hexadecimal numerals (0-9, A-F) [16, 20, 33, 34].
The **permutation encoding** approach is commonly utilized in ordering problems. The individual is represented in this encoding technique with a string of integers that stand for a location in a sequence [16].
In the **value encoding** technique, the individual is represented by a string of values. These values can be real numbers, integers, or characters [11]. This encoding approach may be useful in solving difficulties involving more sophisticated values. In such cases, binary encodings may fail [16].
In **tree encoding**, the individual is represented by a tree of commands. These commands can be in any programming language. This is similar to the representation of repression in tree format [15]. This type of encoding is generally used in evolving programs or expressions [16].

2.2.2 Fitness

The fitness function computes the fitness of individuals. This fitness is problem dependent. A trivial example would be GA used on OneMax problem, where we have individuals in the form of strings of length n. Where each value in the string can be a 0 or 1, thus there are 2^n possible individuals. Our population starts with a population of randomly generated individuals. The goal is to get to a string of all 1s, and we measure fitness of individuals by counting how many 1s they have. The more they have, the fitter they are. Similarly to this problem, we compute fitness for others as well [24].

2.2.3 Selection

The selection procedure is used to derive fitter individuals. The individuals with higher fitness should have a greater chance of being selected than those with lower. Because we want our population to become fitter, and thus to make our solution better. Selection can be created with replacement. This means that fitter individuals will get selected multiple times, or even be recombined with themselves [24]. Some of the better known selections are the roulette wheel, rank, tournament, boltzmann and stochastic universal sampling. We will now explain some of these we used in detail.

Roulette wheel selection works in the following way [32]:

1. Calculate the probability of selection by dividing the individuals' fitness with the population's fitness (sum of all fitness values of all individuals in the population).
2. Divide the roulette wheel into sections based on the probability computed in the previous step.
3. Spin the wheel n times (Here n is the population size). The sector on which the roulette pointer stops is selected, so the individual that corresponds to that sector goes to the next steps of GA. (crossover and mutation)

Rank selection works like this: First, we sort the population by fitness values of individuals. Then we assign the ranks in a way where the fittest individual gets rank n, and the least fit gets rank 1. Then the ranks are assigned to all other individuals linearly according to their fitness. Even if the fitness of 2 different individuals are the same, they get unique ranks [32]. Then, similarly to the roulette wheel selection, we set the ranks on to a roulette wheel, and spin n times, to select the individuals for next steps in GA.

We would like to note here that rank-based selection helps in preventing premature convergence caused by exceptionally fit individuals. It does this because the fittest individuals are always assigned the same selection probability, regardless of their fitness. But, because the best individuals are not that different from others in the population, this strategy may result in delayed convergence [31].

Tournament selection is one of the most popular selection methods, due to ease of implementation and efficiency, say the authors of the paper: [32].

In this selection method, arbitrary n individuals are selected in the following way:

1. The population gets shuffled.
2. Some m amount of individuals (tournament size) is picked from the population.
3. From those, m the fittest individual is picked for next steps in GA.
4. Then we simply repeat this procedure n times, to get n individuals.

2.2.4 Crossover

The crossover method is used to combine the genetic information of 2 or more individuals. Some of the better known crossover procedures are single-point, two-point, k-point, uniform, order, partially matched, shuffle, reduced surrogate, cycle, and precedence preserving crossovers [16]. In the diagrams below we used only 1s and 2s to represent the values of strings, for easier comprehension, in reality we would have some bits, or other sort of encoding we have talked about previously. Now we will further explain the crossovers we used in our implementation:

Single point crossover splits the genomes of 2 individual in half and crosses them like in the figure 4 below, effectively making 2 more individuals [16].

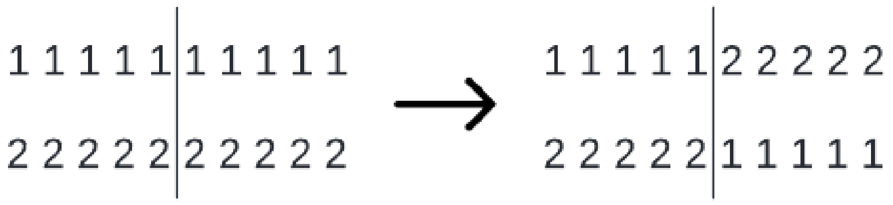

Figure 4: Single point crossover

Two point and K-point crossovers split the individuals' genomes in 2 or more points. Then, as previously, crosses them and creates "offspring". We can see the procedure in the figure 5 below.

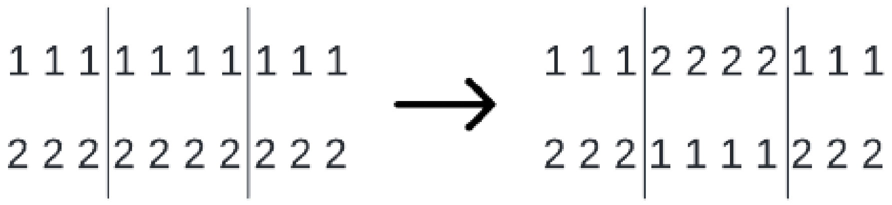

Figure 5: Two point crossover

Similarly, we would have more than 2 of this splitting points with the k-point crossover. Logic is the same [16].
Uniform crossover splits the genomes of 2 individuals on every value in the string. For each value, we toss a coin to decide if we will swap the values of individuals genomes. In the image 6 below, we can see the swap happening with a perfectly fair coin toss, for easier explanation, so every second value gets swapped. In a real scenario, some of the 1s and 2s wouldn't be changed [16].

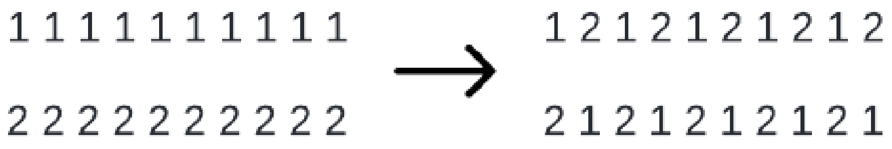

Figure 6: Uniform crossover

2.2.5 Mutation

Mutation is a method used to keep genetic diversity in the population. Some of the better known mutations are displacement, scramble and simple inversion. We will now proceed with the explanation of the displacement mutation as it is the only one we implemented, thus it's the only one relevant to this topic [16].
Displacement mutation displaces a substring of the individuals' genome. The place of displacement is randomly chosen, under constraints that the displaced genome is still valid after the operation. There are 2 other variants of displacement mutation which we would like to mention. The exchange and insertion mutations. Read more about them in this article [16].

3 Implementation

3.1 Problem definition

There is a global problem regarding the number of vehicles and the state of existing road network infrastructure [36]. Every year there are more and more vehicles, and the current infrastructure cannot keep up with that, resulting in more frequent traffic congestion [2, 36]. Road networks are essential for the economic growth of every country [30]. It is necessary to make strategic expansion, adequate maintenance, and optimization of these networks to guarantee an efficient connection within a geographic region. We attempted to replicate this problem by simulating it on the traffic network of Koper. As we previously mentioned, for such problems, it is difficult to find a concrete solution to, since it takes a long time, so we used genetic algorithms to optimize the existing traffic network of Koper [37] and decrease congestion.
For the implementation of both the simulator and the genetic algorithm, we used Java programming language [3].

3.2 Simulator

The simulator we used in our optimization works in the following way when executed. First, it loads up all the intersections, from a provided JSON file [9,10]. All intersections are loaded with their x and y locations, the data that keeps track of which road is the main road, and which are secondary, and most importantly the type of the intersection. This type can be 1, 2 or 3, a normal intersection, semaphore controlled intersection or a roundabout, respectively. Additionally, there are the type 0, which are the parkings, and the type 5 which are the vehicle emitters (more on them later). Then it loads the road data, with which the simulator constructs the road network. It does so by connecting the intersections in a way that checks which roads are incoming, and which are outgoing, in order to make sure the network is connected properly. We should mention here that each road has its own speed, so vehicles that go over them have that set speed, and that each type of intersection has its own implementation.
Now when the whole road network is set up, the simulator starts emitting the vehicles, this is where the vehicle emitters (type 5) come in. They calculate a random shortest

route from the emitting point to some of the parkings (type 0) and emit vehicles across the city in a way that follows real world Koper traffic. We measured the amount of vehicles that passed in certain locations (where emitters emit from) and simulated daily traffic of Koper. On the figure 7 below, you can see what the simulator looks like with the GUI on (graphical user interface = GUI). The circles (vertexes) on the map are the intersections, emitters, and the parkins (color depicts which one it is, and what type). The lines are the roads and the blue squares are the vehicles.

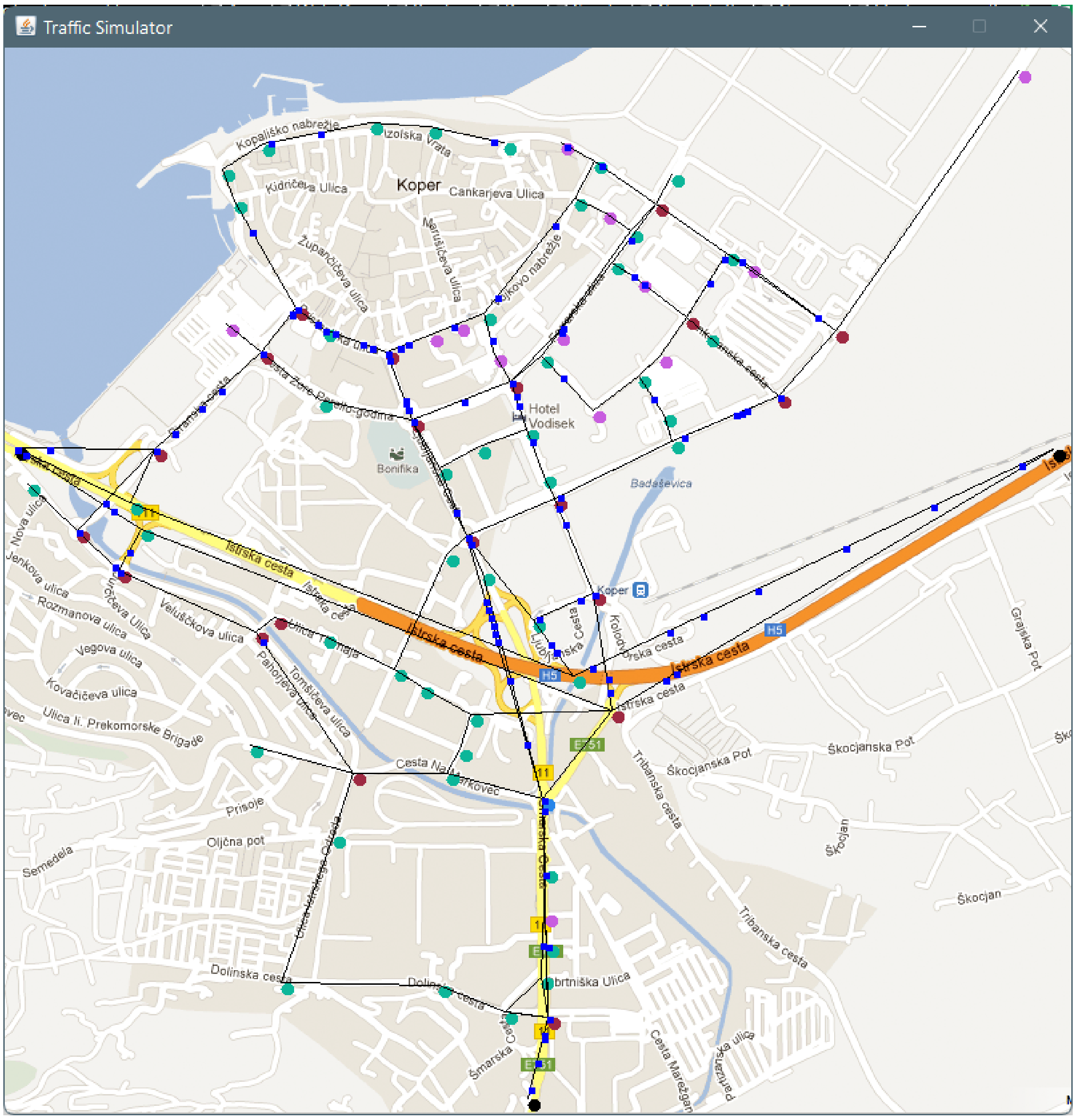

Figure 7: Traffic simulator

Lastly, we should mention here a couple of things. The simulator is sped up, so a whole simulation takes about 3-5 minutes without a GUI, and with the GUI it depends on the hardware of the machine you are running it on. One simulation simulates one entire 24-hour day. Upon completion, it returns the amount of seconds all vehicles spent on the intersections on average. (amount of time spent on intersections / amount of cars)

3.3 Genetic algorithm

Before we dive deep into the explanation of each specific package and class, we would like to briefly explain how we implemented the GA. In the figure 8 below, we can see the algorithms of our Implementation.

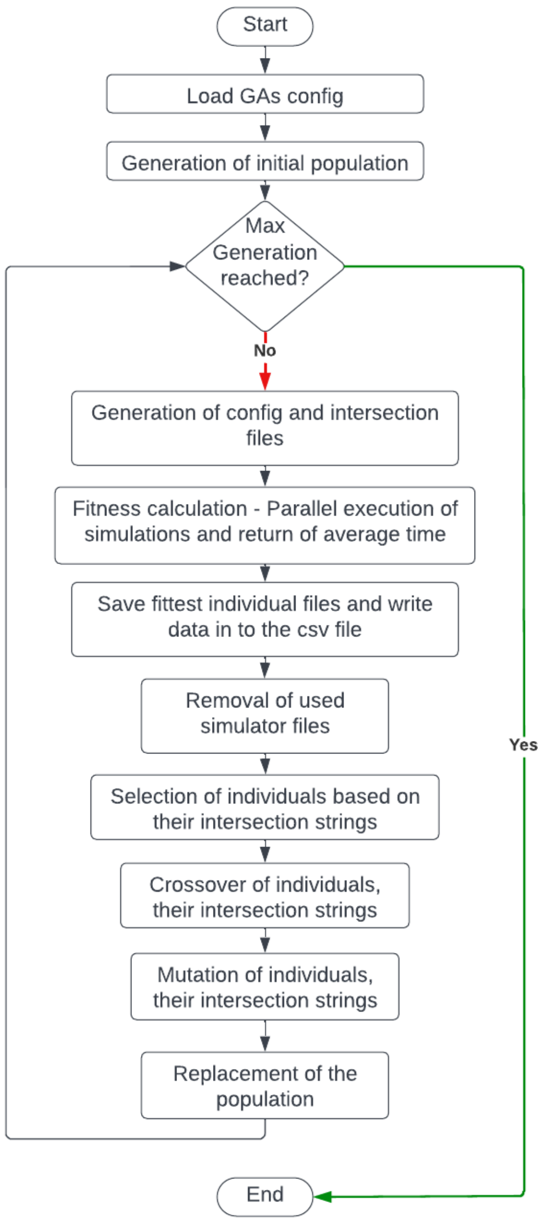
Start
Load GAs config
Generation of initial population
Max Generation reached?
No
Yes
Generation of config and intersection files
Fitness calculation - Parallel execution of simulations and return of average time
Save fittest individual files and write data in to the csv file
Removal of used simulator files
Selection of individuals based on their intersection strings
Crossover of individuals, their intersection strings
Mutation of individuals, their intersection strings
Replacement of the population
End

Figure 8: Implemented genetic algorithm

In the implementation of the GA, we have 3 packages which separate the logic of the project. The algorithms, data, and entity packages.

3.3.1 Data package

Data package is used for reading and writing to memory. In it, we have four classes Config, ConfigData, DataManager, and IntersectionData. We should mention here that there are two configs. One is for the simulators, and the other is for the GA.

- **Config class** is the config for the genetic algorithm, it stores the information for population size, number of threads used for execution, maximum generation for GAs stop condition, mutation probability for the individuals' genomes, simulation speed of individual simulations that are executed, seed of simulation vehicles, the period of time which decides how long the simulation will run, and which genetic algorithm will be used for optimization.
  Furthermore, this class is used as a template when reading a JSON file from disk (in this case, config.json for the GA). This is the first thing we do, as the information previously mentions is critical for any part of this program.

- **ConfigData class**, similarly to the Config class, stores some information, but this time for the simulator itself. This class keeps the simulation speed, the seed of the simulations vehicles, and the simulation time (all 3 are saved in both of the classes).
  This class is used as a template when reading from a base config.json file for the simulator, or when writing to generation specific config files for the simulators. We will explain more about this file writing (saving to disk) later.

- **IntersectionData class** is again a template class used for saving information of each intersection. It keeps the information for the type of the intersection, parking, or emitter (0,1,2,3 and 5 as previously mentioned), the data about main and non-main roads, and the x and y coordinates.

- **DataManager** is the class that has the most logic in this package. It handles the files saving process (writing to disk) in this project.
  Its 4 most important features are:

  - *population_write* method, which saves (writes to disk) the config file of a simulator for a specific generation.
  - *individual_write* method, that saves (writes to disk) the intersection data of a provided individual into a JSON file.

  - *delete_generation_files* method, which removes all executed (completed situations) simulation files (more about execution later). It removes all except the ones of the fittest individuals in the population, so we can run the simulation later and see what GA came up with.
  - *generation_csv_write* method, that writes the population size, algorithm used, mutation chance, name of the fittest individual and their fitness into a CSV file.

3.3.2 Entity package

The entity package contains most of the functionality, and logic in the project. It has the following classes: Main, Optimization, Population, Individual, Executor, Tuple, IndividualComparator and an ENUM Intersection (binary representation of intersections). This package contains the gist of the project, and in it our GA is used, simulations are executed, configs are read and parsed, population and individuals are defined as well as the executors. Before we proceed, it is important to mention that we have two different lists of values. One is a list of complete intersection information (x,y, road priority and type), used when reading from or writing to disk (memory), and the other is a list of simple Intersection ENUM values, used for changing Intersection types. We will now further explain what each class is used for and how it operates.

- **Main class** is our starting point of execution. It reads the program arguments, and the provided config. Then it parses that information and initializes the population, the GA which we will use, and the optimization (more on them later). Finally, the instance of the optimization, we previously created, is started, and our procedure starts.
- **Intersection ENUM** is a binary representation of our 3 types of intersection. Namely, the roundabout, regular intersection and the semaphore controlled intersection. This class is basically one of these three values: "BASIC", "SEMAPHORE" or "ROUNDABOUT", but represented in a binary format.
- **Individual class** is used for storing the necessary information of each individual (our Individuals are a bit more complex than chromosomes). This information includes a list of intersections, individual's fitness, paths to the generation config and the individuals' intersection JSON file, and it's name (i.e., 0_1, 3_24, where the 0 and 3 before the underscore are the generation counts, and the numbers after the underscore are the individual counts, forming an individual name, used when writing the fittest individual to CSV, and when separating Individuals simulation files into directories). Now we'll explain the methods in this class:

  - *Individual constructor methods* are used for creating individuals. There are two of them, and depending on the arguments we either create a completely new individual with a random intersection list, or we make a new individual and assign the intersection list to it. We have this because of the GAs (figure 8, depicts the initial population and the later ones).
  - *randomize_intersection method* creates a new list of intersections and randomly fills it up with the 3 basic intersection values we previously mentioned ("BASIC", "SEMAPHORE" or "ROUNDABOUT"). Used for the creation of the initial population.
  - *type_converter method* is used for converting "BASIC", "SEMAPHORE" or "ROUNDABOUT" basic binary values into numbers, 1,2,3 respectively.
  - *initialise method* is used for saving the information of the entire intersection list. It iterates the list and to each intersection a new type is assigned from the intersection ENUM list we created randomly, or from the intersection ENUM list GA provided, through the constructor. Here we use the *type_converter method* for each of these values in order to save numbers instead of bits.
  - *other methods* such as prints, getters, and setters for some values in the class are also present, but they are trivial as they are self-explanatory (getters get the value of a variable, and setters set the values of variables).

- **Executor class** is used for starting a single simulation. It extends JAVAs class Thread, which enables us to run multiple simulations in parallel (more parallel execution in the Optimization class) [3]. Two main methods that must be explained are:

  - *initialise method* takes as a parameter an individual which it has to execute. It saves the paths needed for execution (the generation specific config.json path and the individuals intersection.json path).
  - *run method* creates a system specific command and runs it. A command looks like the following for windows operating systems:
    ```
    cd "C:\Users\G6\IdeaProjects\diploma\diploma\simulator" &&
    java -jar Simulator.jar false ..\generations\0\config.json
    ..\generations\0\0_3\intersections.json
    ```
    The `cd` and the path after moves the program to the required directory. Then it starts the `Simulator.jar` with `java` and the following arguments. The `false` stands for no GUI, and the next two paths are used to designate the locations of the config for the simulator and the intersection.json to load and run the simulator

on. When the simulator completes the simulation, it returns the fitness and assigns it to the Individual which was provided to it in the *initialise method.*

- **Population class** stores the necessary information about the population. All of the individuals (stored in a vector), the generation count, the population size (provided by the config loaded in Main), the list of instances (objects) of class IntersectionData, an instance of ConfigData class, and it's generation specific copy as well as the path to the generation specific config.json. This class is used for setting up the population for execution. It contains the following methods:
  - *Population constructor methods*, which are used for creating populations. Similarly to the Individual class, here we have 2 methods that are called the same but are used in different ways. One is used when creating the first population, and the other is used for overwriting the existing population with a new one (for the GA).
  - *initialiseGeneration method* checks if the population already exists. If it **does not** exist, the method creates new individuals. If it does exist, it sets up the individual names, their new generation config and intersection paths. Furthermore, it calls the *initialise* method for each individual in the population. It uses an instance of DataManager class to save all individuals in the memory (writes all individuals to disk), and prepares them for execution.
  - *getFittestIndividualSORTED & getFittestIndividual methods* either first sorts the population and then returns the fittest individual, or just returns the fittest individual since the population was previously sorted.
  - *loadIntersections & loadConfig methods* load the intersections and config data from JSON files on disk. They both use template classes, we previously mentioned, IntersectionData and ConfigData respectively.
  - *other methods* such as prints, getters, and setters for some values in the class are also present, but they are trivial as they are self-explanatory.
- **Optimization class** when instantiated sets the parsed parameters provided by the Main class. These are the same mandatory config variables we previously mention in the Data package. This class is used for looping the GA until we reach the MaxGeneration, our stop condition.

This class has two methods:

- *Optimization constructor method* sets up the required variables loaded in Main. It initializes a new instance of a DataManager and an instance of a blocking queue (in laymen terms, a blocking queue is a queue that works properly when using parallelism in our programs) [3].
- *Start method*, first initializes everything we need for parallel execution. (i.e., blocking queue, executor service etc.) Then the main part of the program starts. We loop until the MaxGeneration is reached. In every iteration (repetition) of the loop, the following happens in order:
    1. The population is initialized (*initialiseGeneration* is called).
    2. All individuals in the population are added into the blocking queue we previously defined.
    3. Now, until this queue is completely empty, we assign the individuals to the executors and then execute them in batches.
    4. Join the Threads (Executors), meaning we wait until all of them finish.
    5. Sort the population by fitness, write the fittest individual into a CSV file, and remove files of all other individuals from the disk.
    6. Use the GAs method, *select*.
    7. Use the GAs method, *crossover*
    8. Use the GAs method, *mutate*

- **Tuple class** is a simple class used for storing two instances (objects) of the Individual class. We need this for some of the GAs methods.
- **IndividualComparator class** is used for comparing two instances of the Individual class, based on their fitness values.

3.3.3 Algorithms package

This package stores all of the genetic algorithms we have implemented. We would like to define a variable here, for easier use and explanation. Let n be the size of the population. These are the classes we implemented:

- **Abstract GA class** is used as a template for other classes. All other GAs extend this one. It contains the basic 3 methods, *select crossover*, and *mutate*. Read more about abstraction in [3].
- **GenericGA** which has implemented the 3 methods like this:

  - *select* sorts the population by using an instance of the IndividualComparator class, selects the upper half (the fitter half, $n/2$) and deletes the lower half.
  - *crossover* uses single point crossover. It iterates the newly selected population 2 at a time. It splits the lists of Intersections of both individuals, in half and crosses them over. So the 1st half of the 1st individual and the 2nd half of the 2nd individual are merged into a single new list for the first child, and the opposite is done for the second child. So the 1st half of the 2nd individual and the 2nd half of the 1st individual are merged into a single new list for the second child.
    Since we have to return 2 instances (objects) of the Individual class, here we use the Tuple class to return both of the children, and add them to the new population.
  - *mutate* randomly changes some intersections in the list of Intersections. It does this by iterating through the list, and at every iteration it generates a local random number. If that number is less than or equal to the mutation chance, we flip the value to one of the other two values (since we have 3, "BASIC", "SEMAPHORE" or "ROUNDABOUT").

- **TournamentGA**
  - *select* works in the following way:
    1. Define the new population, which is the size of $n/2$ (after crossover, it will have the same size as before). Define a new vector of type Individual (vector is a data structure of flexible size) which is the size of the tournament. Let this size be some t. In our case $t = 5$.
    2. Loop until we reach the $n/2$.
    3. In each iteration, we shuffle the population and pick t individuals. After that, we pick the fittest of the those and insert it into the new population, and remove it from the original population, so we won't have duplicates.
  - *crossover* is the same as in **GenericGA**.
  - *mutate* is the same as in **GenericGA**.
- **Tournament2PointGA**
  - *select* is the same as in **TournamentGA**.
  - *crossover* uses the two point crossover. It iterates the newly selected population 2 at a time. This method splits the lists of Intersections of both

individuals, in 3 peaces, and crosses them over, similarly to the single point crossover. Refer to our explanation of 2 point crossover and the diagram 5 for a deeper explanation.

  - *mutate* is the same as in **GenericGA**.

- **TournamentUniformGA**
  - *select* is the same as in **TournamentGA**.
  - *crossover* uses uniform crossover. It iterates the population 2 at a time. For both, individuals it again iterates their Intersection lists, and for each Intersection a random number is rolled between 0 and 1. If this number is less than 0.5 then Intersections are swapped. Thus resulting in two new Individuals, as before. Refer to the diagram 6 for a deeper explanation.
  - *mutate* is the same as in **GenericGA**.

4 Results

In this section, we will talk about the results of the executed simulations. We ran these on a university server with rough specifications.
In the figure 9 we can see the progress of various genetic algorithm implementations in traffic network optimization. On the $y - axis$ we can see the fitness of the fittest individual. On the $x - axis$ we can see the generation count. The fitter the individual is, the lower is its fitness values. Hence, the graph has decreasing values as the GA explores more optimal solutions. The occasional moving up of the fitness is caused by the mutations. Furthermore, we can see that the most optimal solution was found by the TournamentUniformGA, depicted with light blue color in the figure 9. Since it reached the lowest fitness value compared to other GAs.

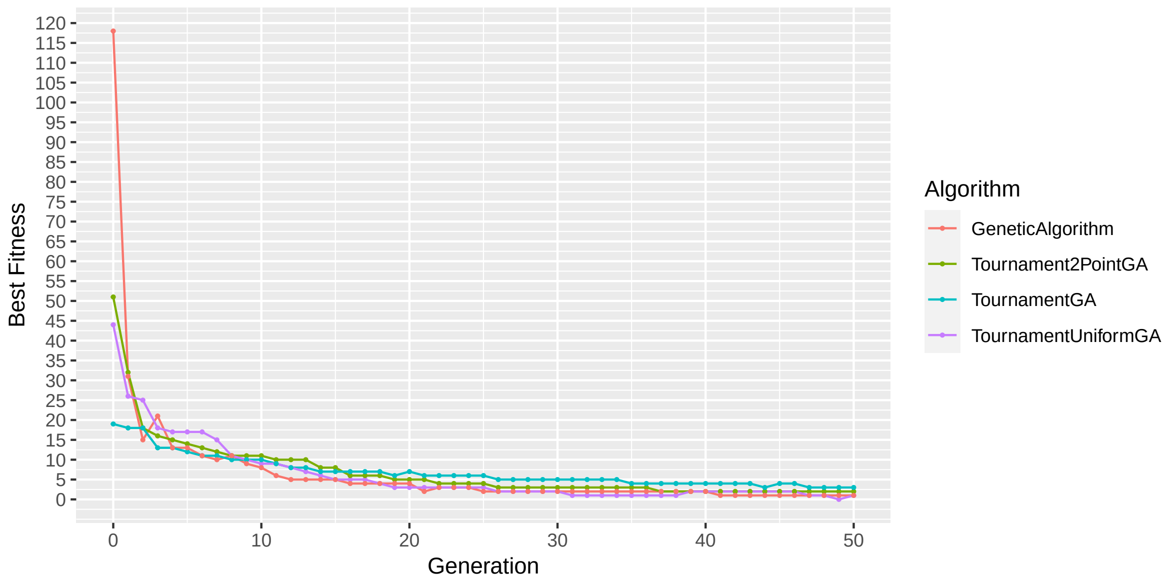

Figure 9: GenericGA graph

5 Discussion

In the results, we saw that the road network can be optimized. Now the further steps would be to make the simulator more realistic, and to make it, so any road network can be imported and optimized, one of possible implementations is using the OpenStreetMap [29]. This would require a lot of work, as many road networks don't have only 3 types of simple intersections, but others as well with more incoming and outgoing roads. This is where GA would also have to be expanded, by adding any new intersection types to the simulator, we would also have to add them to the GA. Additionally, we could also expand the algorithm to multi-criteria optimization, and add rules to which types can be converted into others and at what cost, financially, logistically and in how much time. To expand, we could also research the mutation factors, and implement more combinations of different GAs to gain a better vision of their comparisons.

6 Povzetek naloge v slovenskem jeziku

Ta članek smo začeli s kratko klasifikacijo problemov glede na to, kako težko jih je rešiti. Nato smo razložili, kako za določene probleme obstajajo tehnike in algoritmi, ki nam lahko pomagajo da pridemo do optimalnejše rešitve. To so namreč hevristične funkcije, ki smo jih pojasnili. Nato smo opisali genetske algoritme, saj so podnabor hevrističnih funkcij, in ponazorili, kako delujejo.
Definirali smo problem in kako ga bomo poskušali rešiti. Nato smo dodatno pojasnili, kako deluje simulator na podlagi agentov, in kako ga uporabljamo v genetskem algoritmu za simulacijo več primerkov različnih cestnih omrežij.
Podrobno smo razložili implementacijo genetskega algoritma in kako optimizira cestno omrežje. Nato smo zaključili z rezultati večkratnih izvedb različnih variacij GA.
Na koncu smo zaključili z razpravo, v kateri smo razložili, kaj bi še lahko naredili za izboljšavo simulatorja in genetskih algoritmov ter kako lahko še razširimo celoten projekt.

7 Bibliography

[1] EHL Aarts. A stochastic approach to combinatorial optimization and neural computing. *Simulated Annealing and Boltzmann Machines*, 1989. *(Cited on page 3.)*

[2] Naeem Abbas, Muhammad Tayyab, and M Tahir Qadri. Real time traffic density count using image processing. *International Journal of Computer Applications*, 83(9):16–19, 2013. *(Cited on pages II, III and 12.)*

[3] Ken Arnold, James Gosling, and David Holmes. *The Java programming language.* Addison Wesley Professional, 2005. *(Cited on pages 12, 17 and 19.)*

[4] Mario Bunge. A general black box theory. *Philosophy of Science*, 30(4):346–358, 1963. *(Cited on page 3.)*

[5] Gaurav Dhiman and Vijay Kumar. Spotted hyena optimizer: a novel bio-inspired based metaheuristic technique for engineering applications. In *Spotted hyena optimizer: a novel bio-inspired based metaheuristic technique for engineering applications*, volume 114, pages 48–70. Elsevier, 2017. *(Cited on page 4.)*

[6] Gaurav Dhiman and Vijay Kumar. Emperor penguin optimizer: a bio-inspired algorithm for engineering problems. *Knowledge-Based Systems*, 159:20–50, 2018. *(Cited on page 4.)*

[7] Gaurav Dhiman and Vijay Kumar. Seagull optimization algorithm: Theory and its applications for large-scale industrial engineering problems. *Knowledge-based systems*, 165:169–196, 2019. *(Cited on page 4.)*

[8] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization - artificial ants as a computational intelligence technique. *IEEE Comput Intell Mag*, 2006(28–39), 2006. *(Cited on page 4.)*

[9] E.C.M.A. The json data interchange format. *(Cited on page 12.)*

[10] Ed. Errata Exist, T. Bray. Internet engineering task force (ietf). *The Format*, 2014-03. *(Cited on page 12.)*

[11] B. Fox and M. McMahon. Genetic operators for sequencing problems. In G. Rawlins, editor, *Foundations of Genetic Algorithms*, page 284–300. Ed Morgan Kaufmann and San Mateo, C.A., 1991. *(Cited on page 8.)*

[12] F Glover and M Laguna. Tabu search kluwer academic. *Boston, Texas*, 1997. *(Cited on page 3.)*

[13] H.Iima Guoyong Shi and N. Sannomiya. A new encoding scheme for solving job shop problems by genetic algorithm. In *Proceedings of 35th IEEE Conference on Decision and Control*, page 4395–4400 4, Kobe, Japan, 1996. *(Cited on page 8.)*

[14] W Hämäläinen. Class np, np-complete, and np-hard problems, 2006. *(Cited on pages 1 and 2.)*

[15] K. Jebari. Selection methods for genetic algorithms. abdelmalek essaâdi university. *International Journal of Emerging Sciences*, 3(4):333–344, 2013. *(Cited on page 8.)*

[16] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*, 80(5):8091–8126, 2021. *(Cited on pages 4, 5, 8, 10 and 11.)*

[17] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. IEEE, 1995. *(Cited on page 4.)*

[18] D. Kreher and D. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search.* CRC Press, Boca Raton, FL, 1998. *(Cited on page 3.)*

[19] Donald L Kreher and Douglas R Stinson. *Combinatorial algorithms: generation, enumeration, and search.* CRC press, 2020. *(Cited on page 3.)*

[20] A. Kumar. Encoding schemes in genetic algorithm. *Int J Adv Res IT Eng*, 2(3):1–7, 2013. *(Cited on page 8.)*

[21] Vijay Kumar and Dinesh Kumar. An astrophysics-inspired grey wolf algorithm for numerical optimization and its application to engineering design problems. *Advances in Engineering Software*, 112:231–254, 2017. *(Cited on page 4.)*

[22] Joon-Yong Lee, Min-Soeng Kim, Cheol-Taek Kim, and Ju-Jang Lee. Study on encoding schemes in compact genetic algorithm for the continuous numerical problems,sice. *Annual Conference*, page 2694–2699, 2007. *(Cited on page 8.)*

[23] D Levy and M Newborn. How computers play chess esp, 1991. *(Cited on page 3.)*

[24] John McCall. Genetic algorithms for modelling and optimisation. *Journal of computational and Applied Mathematics*, 184(1):205–222, 2005. *(Cited on pages 5, 7, 8 and 9.)*

[25] Z. Michalewicz. *Genetic algorithms + data structures = evolution programs.* Springer-Verlag, New York, 1992. *(Cited on page 4.)*

[26] Seyedali Mirjalili. Genetic algorithm. In *Evolutionary algorithms and neural networks*, pages 43–55. Springer, 2019. *(Cited on pages II and III.)*

[27] Melanie Mitchell. *An introduction to genetic algorithms.* MIT press, 1998. *(Cited on pages 3 and 5.)*

[28] B John Oommen and Luis G Rueda. A formal analysis of why heuristic functions work. *Artificial Intelligence*, 164(1-2):1–22, 2005. *(Cited on page 3.)*

[29] OpenStreetMaps. Open street maps. *(Cited on page 23.)*

[30] Cesar AV Queiroz and Surhid Gautam. *Road infrastructure and economic development: some diagnostic indicators*, volume 921. World Bank Publications, 1992. *(Cited on pages II, III and 12.)*

[31] Noraini Mohd Razali, John Geraghty, et al. Genetic algorithm performance with different selection strategies in solving tsp. In *Proceedings of the world congress on engineering*, volume 2, pages 1–6. International Association of Engineers Hong Kong, China, 2011. *(Cited on page 9.)*

[32] Anupriya Shukla, Hari Mohan Pandey, and Deepti Mehrotra. Comparative review of selection techniques in genetic algorithm. In *2015 international conference on futuristic trends on computational analysis and knowledge management (ABLAZE)*, pages 515–519. IEEE, 2015. *(Cited on page 9.)*

[33] S.N. Sivanandam and S.N. Deepa. *Introduction to genetic algorithm.* Springer-Verlag, Berlin Heidelberg, 1st edn edition, 2008. *(Cited on page 8.)*

[34] Lima S.J.A. and Araújo S.A. A new binary encoding scheme in genetic algorithm for solving the capacitated vehicle routing problem. In Korošec P., Melab N., and Talbi EG, editors, *Bioinspired Optimization Methods and Their Applications. BIOMA 2018. Lecture notes in computer science*, volume 10835. Springer, Cham, 2018. *(Cited on page 8.)*

[35] Larry Stockmeyer. Classifying the computational complexity of problems. *The journal of symbolic logic*, 52(1):1–43, 1987. *(Cited on pages 1 and 2.)*

[36] Galina Tokunova and Marlena Rajczyk. Smart technologies in development of urban agglomerations (case study of st. petersburg transport infrastructure). *Transportation Research Procedia*, 50:681–688, 2020. *(Cited on pages II, III and 12.)*

[37] Wikipedia. Koper, slovenia. *(Cited on pages II, III and 12.)*

[38] Stelios H Zanakis and James R Evans. Heuristic "optimization": Why, when, and how to use it. *Interfaces*, 11(5):84–91, 1981. *(Cited on page 3.)*