Automated testing for self-driving cars using real-world roads

Version of August 16, 2022

Bart Roseboom

Automated testing for self-driving cars using real-world roads

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Bart Roseboom
born in Capelle aan den IJssel, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Automated testing for self-driving cars using real-world roads

Author: Bart Roseboom
Student id: 5181917
Email: b.roseboom@student.tudelft.nl

Abstract

Cyber-physical systems are complex systems constructed from different independent parts. A self-driving car is an example of a cyber-physical system where independent parts have to come together in order to result in a car that is able to drive by itself. The main challenge is finding failures within the interactions between the independent parts of the self-driving system. In this paper, we present a novel algorithm REWOSA, in order to detect these faults within the self-driving software. With the use of real-world roads extracted from Google Maps combined with a multi-objective genetic algorithm, we develop a new way to generate roads for testing self-driving cars. We evaluate this algorithm against a state-of-the-art multi-objective genetic algorithm using randomly generated roads using two different setups. Our results show that REWOSA is able to generate more failures than the baseline on both the setups, as well as create more complex roads. In return, REWOSA does create a large overhead due to the complexity of the real-world roads. However, this overhead is justifiable as we can detect more faults with the more complex real-world roads.

Thesis Committee:

Chair:	Prof. Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
University supervisor:	Dr. A. Panichella, Faculty EEMCS, TU Delft
Committee Member:	Dr. C.C.S. Liem, Faculty EEMCS, TU Delft

Preface

First I would like to thank Annibale Panichella, my supervisor for this thesis, as he was able to help me pick an interesting research topic as well as provide the right assistance I need. Whether I needed help with something or feedback on my progress, he was able to point me in the right direction or bring me in contact with someone that could help me.

Secondly I would like to thank Pouria Derakhshanfar for all the tips and help with reviewing of the progress on the report.

Next I would like to thank Cynthia Liem, as she approached me first about a potential thesis topic I could do with her and Annibale as supervisors. She brought me in contact with Annibale and from there on out I was able to create this thesis.

I would also like to thank Alessio Gambi from Passau University, he was a lot of help in setting up the simulation engine and making it work with the software.

On top of that, I would like to thank Christian Birchler from Passau University, he was able to provide me with the algorithm that gathered the results for the properties of the roads.

Lastly I would like to thank my family for always supporting and pushing me to the point where I was able to actually finish the thesis.

Bart Roseboom
Delft, the Netherlands
August 16, 2022

Contents

List of Figures

Chapter 1

Introduction

A self-driving car, or an autonomous vehicle, is a car that is able to drive with little to no human input. With the use of sensors and cameras, the car can get an idea of the surroundings and uses that to make decisions on how to drive. Some components in a self-driving car system, which can take over human control, are *Adaptive Cruise Control*, where the car has control over the engine and brake power in order to maintain and fluctuate speed, *Parking Assistant*, where the car has control over steering the vehicle in the designated parking spot, and *Lane Keeping Assistant*, which makes use of a forward facing camera that scans the road for the lane markers to makes sure the car stays within its lane.

Over the last years, we witnessed a large growth in research and development of self-driving/autonomous vehicles. As an example, Tesla plans full hardware support for self-driving cars [39], and Waymo initiates a ride-share program with self-driving cars [34] However, these developments also showed some negative pointers. The main pointer is the danger of testing self-driving vehicles on real roads. Many recent articles reported the scenarios in which self-driving cars caused both minor accidents [9, 32, 36] and fatal crashes [9, 32]. On top of the fact that testing self-driving cars on roads with real traffic is dangerous and expensive, a study [18] also showed that it is not sufficient enough to assure the safety of cars under test. They state that the cars need to drive multiple millions or even billions of miles in order to provide statistical data confirming the safety of the cars.

A common technique used within the world of Cyber-Physical systems is the use of Hardware-in-the-Loop (HIL) simulations, which provides a platform for testing the complex real-time embedded systems in a safe space. Besides safety, this platform eases the process of testing. For instance, developers can test the system in various scenarios without waiting for specific weather conditions or finding specific road structures [4].

Next to HIL, we also have Software-in-the-Loop (SIL). SIL provides an environment for developers to fully test the software of a real-time system without involving any hardware module.

With the aim of reducing the cost and danger and providing a more efficient way of testing, the use of virtual testing provides a good alternative to real-world testing. The main components of these virtual testing scenarios are models of the hardware and software of the vehicle under test. As the hardware components of self-driving vehicles include cameras and sensors, one way of virtually testing the vehicle is by providing artificially gen-

erated images, or sensor data [4, 41]. Another way of virtually testing self-driving vehicles is by creating real-world-like environments and scenarios within a virtual world [15, 22]. Simulating different scenarios and environments is easier than actually testing them in real life. In real life, the correct place with the desired conditions needs to be found, whereas in simulations, the desired environment is given to a simulation engine, and it is directly available. Nevertheless, creating such virtual environments does not come without its challenges. First, we need to be able to generate the virtual world (*e.g.,* generating roads), its surroundings, the landscape, and weather conditions. Secondly, when generating these virtual worlds, we need to create complex and challenging test scenarios. In other words, we need to create an environment in which the vehicles show weaknesses within the hardware or software implementation.

One way of helping to improve the process of generating these test scenarios is using seeding strategies, which can be defined as using previously gathered knowledge in order to help solve the current testing problem. Various papers showed that using seeding strategies improve the performance of search-based software testing techniques [11, 13, 30].

We aim to create a tool generating various diverse test cases that are able to find out-of-bounds episodes (OBEs) while also keeping the number of invalid test cases low. To achieve this goal, we introduce a novel seeding strategy for generating roads, called `REWOSA`. `REWOSA` uses real-world roads gathered from Google Maps to guide a multi-objective search-based test generation process for finding flaws in the self driving software from BeamNG. Our algorithm is based on FITEST [2], a many-objective test generation algorithm, with the addition of mutation to push the software to failures. `REWOSA` utilizes the data available in Google Maps to improve the road creation process by extracting real-world roads and converting them so that they are usable with a simulation. We then provide an empirical assessment of our new algorithm by comparing it with a basic genetic algorithm as a baseline.

In our algorithm, the roads are represented by a list of points, consisting of an X and Y coordinate, within a 2D grid. The way the points are generated is done in two different ways, one in which we generate random points, using the Deepjanus seed generator, included in the SBST21 code package [1]. The other way uses roads gathered from Google Maps and transforms the real world coordinates into points within the 2D grid. We then keep a list of all the test cases that do not generate an OBE, and that are not dominated by other test cases that do not generate an OBE. The test cases in this list are then randomly used to be mutated to create a test case that finds an OBE.

In order to assess our approach we compare our `REWOSA` to a genetic algorithm as baseline, using two different setups. We use the BeamNG.research simulator with the BeamNG AI self driving software to assess both algorithms on. We found that `REWOSA` was able to detect more then 2 times the amount of faults on average and slightly more diverse faults than the baseline. On top of that, we saw that `REWOSA` creates more complex road than the baseline. This results in a trade-off between complexity and overhead, where the higher the complexity of the road, to bigger the overhead will be. But, this bigger overhead of the real-world roads is worth it, as the real-world roads are able to detect more faults.

With this thesis we provide the following contributions:

[1] `https://sbst21.github.io/`

- We introduce a new algorithm called REWOSA, where we use multi-objective search to highlight flaws within the self driving software of BeamNG.
- We implemented our algorithm for the assessment.
- We evaluate REWOSA by comparing it with a genetic algorithm as a baseline, using the BeamNG.research simulator with the BeamNG AI self driving software.

The structure of the rest of the thesis is as follows. In section 2 we discuss the background of the research and discuss related work. Section 3 describes our approach to solve the discussed limitations of current solutions. In section 4 we describe our experimental setup. Section 5 displays and explains our results and discuss them. In section 6 we go over the threats to validity and section 7 concludes our thesis.

Chapter 2

Background and Related Work

2.1 Self-driving cars

A self-driving car, also known as an autonomous vehicle, is a car that is able to maneuver around safely with little to no human involvement. The car can drive safely with the use of cameras, sensors, complex algorithms, and even machine learning. With the use of sensors and cameras, the car can create a map of its surroundings. It uses radar sensors to get an idea of where any nearby vehicles are positioned. Lidar (Light detection and ranging) sensors are used to measure the distances to the other vehicles and detect lane markers and the edges of the road. On top of this, it uses cameras to track pedestrians, traffic lights, traffic signs, and other vehicles. The complex algorithms and machine learning algorithms then use all the information from the sensors and cameras as input to create a path for the car to follow. In order to follow this path, these algorithms send commands to the actuators of the car (*e.g.,* acceleration, braking, and steering). With the help of some other build-in rules, the car is able to safely maneuver around in traffic while abiding by traffic laws.

The idea of a self-driving car all started in 1925, when Houdina demonstrated a radio-controlled car, with no one at the steering wheel. This car was able to start its engine and shift gears just through radio impulses sent from a human-controlled antenna in the car that was following the radio-controlled car.
Then in 1969, McCarthy's essay on his idea of a robo-chauffeur [23], let many other researchers to go deeper into the self-driving car area. His idea was that the automatic driver was able to navigate to a destination using camera inputs. That way, the user would be able to type in a destination and make adjustments to the destination while the car is driving towards the original goal.
During the 90s, Pomerleau described in his thesis [33] how neural networks could transform road imagery into steering commands in real-time. He was able to apply his idea in a minivan and drove from the east to the west coast by just controlling the gas and brake pedals.
In the early 2000s, some modern cars were introduced with a self-parking system, which was able to park in most challenging real-world situations with the use of sensors.
In the last couple of years, the biggest improvements have come from using artificial in-

telligence and machine learning. With the help of Machine learning and neural networks, research has been able to train self-driving cars on how to drive [10, 17].

The innovations within the self-driving field have gotten to the point where we can now simulate the behavior of self-driving software within a digital environment. Previous research shows the use of neural networks and artificial intelligence within simulation engines [10, 21]. Not only that, there is research that uses real-world data in order to generate a realistic virtual world simulator to train perception algorithms [22]. Similar to previously done research, the combination of simulation and an AI controlling the car is used within our research to provide an assessment of our new algorithm `REWOSA`.

2.2 Search-based and evolutionary testing

Search-based software testing is applying optimizing search techniques, like genetic algorithms, to solve software testing problems. Evolutionary testing transforms testing objectives into search problems and applies evolutionary computation to solve the problems in order to improve the efficiency and effectiveness of the testing process. In evolutionary computation, we have a population of solutions, which are subject to natural selection and mutation. This idea stems from Charles Darwin and his theory about the natural evolution. He said that over the course of generations, biological organisms evolve based on natural selection, also known as 'survival of the fittest' [37]. This process of natural selection will gradually evolve the population and improve its fitness. Within the software, this fitness refers to a selected fitness function(s) for the algorithm in use, which provides scores for each solution within the population. With these scores, we can compare two solutions with each other and decide which is the better solution. As an example, assume that in a search process, solving a maximization problem (*i.e.,* higher score is better), we have three search objectives (*i.e.,* each individual is evaluated according to three fitness functions). In a population of solutions, we have a solution that has scores of 2, 3 and 5 (for each of the search objectives, respectively) and a second solution that has scores of 1, 2 and 3. Here, we can clearly see that the first solution is better than the second solution, as the respective numbers are all higher in the first solution ($2 > 1$, $3 > 2$, $5 > 3$). This means that solution one dominates solution two. Lets now say that solution one has scores 2, 1, 5 and solution two has scores 3, 2, 1. Now it is a lot harder to say which solution is better, as we don't know how important the respective scores are. As the first two scores are higher for the second solution ($2 < 3$, $1 < 2$), but the third score is higher for solution one ($5 > 1$), both of these solutions are non-dominant and non-dominated. When we get more and more solutions to compare with each other, the number of non-dominated solution will start to grow. This set of non-dominated solutions is also called the Pareto set or the Pareto front.

A genetic algorithm usually contains three main steps: selection, crossover, and mutation [26]. Selection is the process in which chromosomes, or solutions, are selected for reproduction, where the higher their fitness score is, the greater the chance they will be selected. In the crossover step, sub-sequences of two chromosomes, or solutions, are combined to create new solutions. As an example, let us assume that we have a chromosome

with eight bits, and we cut them both after the fifth bit. This means that we combine the five first bits from the first chromosome with the three last bits from the second chromosome, and vice versa, resulting in two new chromosomes. Then, the mutation step is able to flip each bit with a given probability, most of the time a low percentage.

Within our research we also use evolutionary algorithms, in particular the genetic algorithm. However, we slightly changed from the most basic form of the genetic algorithm. We do not make use of the crossover steps, as we are using road points in a 2-dimensional grid. This would make it hard for the 2 sub-sequences of roads to align just right to create new valid roads. We do adopt the selection step, where we only keep the Pareto set and select a single solution that we then mutate. As we do not use crossover, we always use mutation, where we randomly select whether we add, remove or adjust points within the selected solution.

Previous research by McMinn [25] highlights the origin and rise of search-based testing, explaining the explosion in the amount of work on search-based testing in the early 2010s. Harman *et al.* discuss the lack of theoretical and empirical analysis of search-based testing techniques and present a theoretical exploration of the genetic algorithm [16].

2.3 Test case generation for Self-driving cars

When testing the self-driving car systems, it is shown that this can be very dangerous when performed on real-world roads [9, 32, 36]. On top of that, testing the car on all the different types of roads and climates is a time-taking process. Hence, a common practice both in industry and research for self-driving car systems (in general, any cyber-physical system) is utilizing a simulation engine to safely test this type of system. In addition, simulation-based testing provides the opportunity of running more test scenarios virtually and prevents spending time and cost on testing the system in the real world.

The self-driving car system includes many different components that allow the car to drive without the help of a human driver. Previous studies have focused on single aspects of this driving system [2, 4, 15, 20, 38]. Alghodhaifi and Lakshmanan use radar and camera images as input in order to test the Pedestrian Protection System [4]. Duy Son *et al.* developed their own simulation platform and displayed it by testing adaptive cruise control, green wave technology, autonomous valet parking, and double lane change separately [38]. Klück *et al.* test the automated emergency brake system by using a genetic algorithm for test parameter optimization [20]. Gambi *et al.* combine procedural content generation and search-based testing in order to create virtual roads to test the lane-keeping assistant [15]. There have also been investigations done on testing the combination of these features and how they interact. Ben Abdessalem *et al.* created FITEST, which is a search-based test generation algorithm, in order to test the feature interaction between Autonomous Cruise Control, Traffic Sign Recognition, Pedestrian Protection, and Automated Emergency Braking [2].

Moreover, many testing frameworks for self-driving cars have been developed in recent years. Weissnegger *et al.* created a design, simulation, and verification framework named SHARC, which focuses on verifying safety-critical networked embedded systems concern-

ing functional safety [42]. Duy Son *et al.* established a framework for high fidelity vehicle dynamics, sensors, and traffic environment modelling [38]. Negrut *et al.* created a platform called CAVE that helps with low-cost, risk-free, and rapid testing of new state-of-the-art designs, methods and software components for autonomous navigation [28]. Saraoglu *et al.* designed a simulation-based fault injection framework to assess the safety of autonomous driving systems [35]. Abdelhamed *et al.* used a Robot Operating System and a 3D simulator to produce a simulation framework in which it is possible to virtually design, verify, and validate Autonomous Driving and Advanced Driver Assistance Systems features [1]. For our research, we use a framework set up by the Search-based Software Testing (SBST) workshop's tool competition for automatically generating tests for self-driving cars [31]. The framework is used to implement a design for the generation of test cases to test the lane-keeping feature of a self-driving car system.

In addition, several techniques are proposed for constructing the driving roads used for testing vehicles during simulation. Gambi *et al.* proposed a technique that gradually produces road segments and glues them together in order to ensure gapless roads [15]. In our case, we use real-world coordinates, and thus use points to represent the roads for both our novel generator.

A search-based test generation technique aims to convert test generation problems (in any testing level or system) into an optimization problem. Then, it tries to solve the optimization problem using search algorithms. Similar to any optimization problem, the search-based test generation problem can be a single objective [19], multi-objective (contains two or three objectives) [6], or many-objective (contains more than three objectives) [2] optimization problem.

2.3.1 Solution representation

In order to solve an optimization problem, different solutions are required. These solutions can be represented in different ways, depending on the input required for the algorithm or simulator. Within FITEST, each of these solutions is represented by (i) the initial position and speed of the ego car, (ii) the initial position and speed of the leading car, (iii) the initial position, speed, and orientation of the pedestrian, (iv) the position of the traffic sign and (v) the fog degree.

Within our research, we test a different scenario than FITEST, as we only use one car that tries to drive outside a lane. Therefore we do not make use of a second car, a pedestrian, or a traffic sign, and we also do not make use of the fog degree. The input for our simulation is just the road for the car to drive on. The simulator will place the car at the start of this road, standing still.

Interpolation

In order to create a road that can be interpreted by the simulator, we make use of an interpolation over a set of road points. There are a plethora of interpolation techniques one can use, but in our case, we make use of spline interpolation.

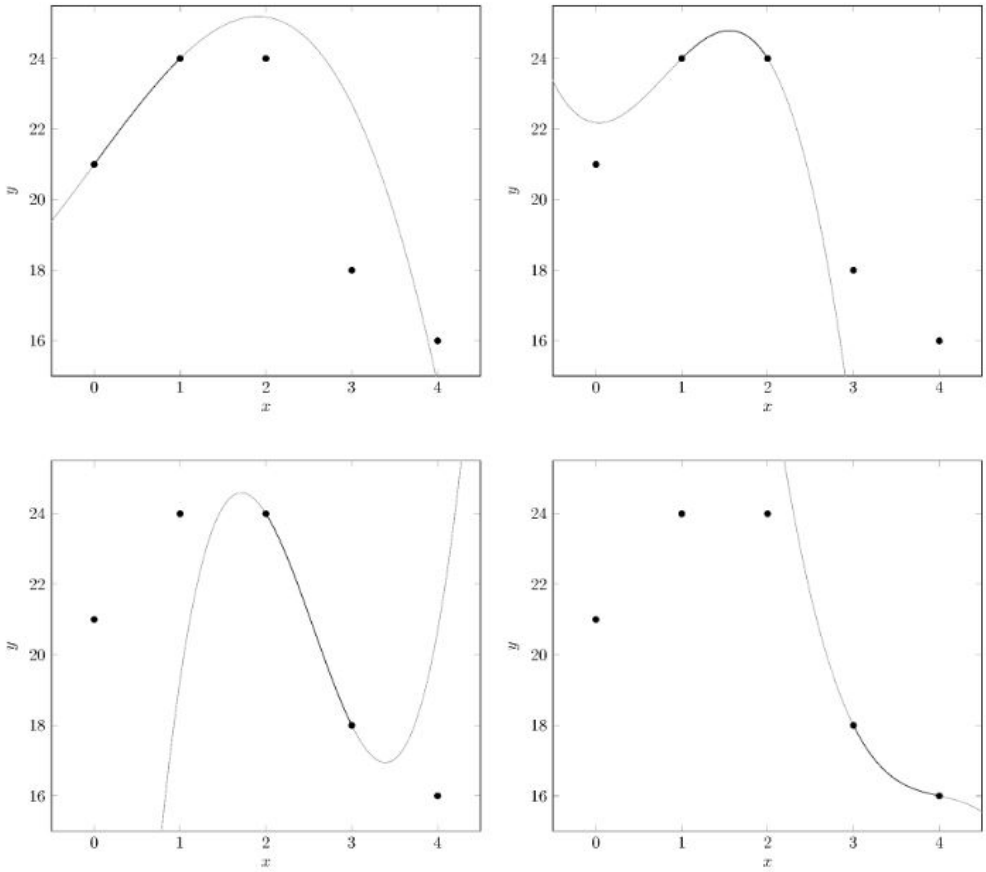

Figure 2.1: The polynomials shown here represent the interpolation function within their given intervals (colored in black). The polynomials functions do not match the points outside of these bounds.[1]

Spline interpolation is an interpolation technique where the function calculating the missing points is a piece-wise polynomial, called a spline. This means that spline interpolation fits low-degree polynomials to a small subset of values instead of fitting one high-degree polynomial between all values. For example, fitting seven cubic polynomials between each of the pairs of eight points instead of fitting one degree-eight polynomial to all the points. Figure 2.1 depicts an example where we can see four polynomials fitted between the four pairs of points, highlighted by the black part of the line, to create the spline together.

To be more specific, we make use of the cubic spline. The fundamental idea behind the cubic spline stems from an engineer's tool to draw a smooth curve through a set of points [24]. At these points, weights are attached such that a strip can be bent around these points to create a smooth bend. The mathematical spline is similar, but the points are numerical data in this case, and the weights are coefficients. Interpolation between these

[1] https://timodenk.com/blog/cubic-spline-interpolation/

points and their corresponding coefficients then makes a line with smooth curves without losing continuity. The cubic spline represents a series of unique cubic polynomials fitted in between the data points. The requirement for the resulting curve is that it has to be a smooth and continuous curve. A mathematical definition of the cubic spline is given as follows [24]:

$$S(x) = \begin{cases} s_1(x) & \text{if } x_1 \leq x < x_2 \\ s_2(x) & \text{if } x_2 \leq x < x_3 \\ & \vdots \\ s_{n-1}(x) & \text{if } x_{n-1} \leq x < x_n \end{cases} \tag{2.1}$$

where s_i is a third degree polynomial defined by:

$$s_i(x) = a_i(x-x_i)^3 + b_i(x-x_i)^2 + c_i(x-x_i) + d_i \tag{2.2}$$

for $i = 1, 2, ..., n-1$.

There are two fundamental derivatives of this formula, which are the first and the second derivative:

$$s_i'(x) = 3a_i(x-x_i)^2 + 2b_i(x-x_i) + c_i \tag{2.3}$$

$$s_i''(x) = 6a_i(x-x_i) + 2b_i \tag{2.4}$$

for $i = 1, 2, ..., n-1$.

The cubic spline must conform to four conditions:

- The piecewise function $S(x)$ will interpolate all data points
- $S(x)$ will be continuous on the interval $[x_1, x_n]$
- $S'(x)$ will be continuous on the interval $[x_1, x_n]$
- $S''(x)$ will be continuous on the interval $[x_1, x_n]$

An example of a cubic spline is shown in figure 2.2. Here, we can see a smooth and continuous line between all the data points. As the line is smooth and follows through all the data points we can see that it is a valid cubic spline.

However, there are some downsides to using the cubic spline interpolation. If the data point values are large and have a big distance between them, the interpolation will become less accurate and can result in incorrect interpolations by multiple orders of magnitude. However, since we use a 950 by 950 grid, and thereby the distance between the points will not become large enough to give large incorrect interpolations, the limitation of cubic spline interpolation does not have any effect in our case.

As another limitation, extrapolation is not possible in the cubic spline interpolation. In the case of predicting points outside of the used data points (*i.e.,* before the first or after the

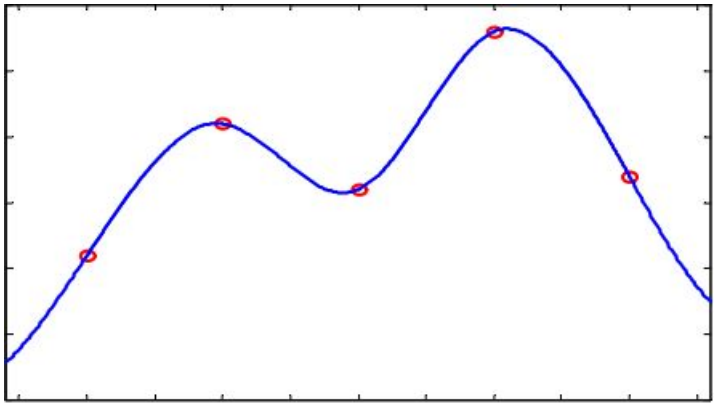

Figure 2.2: An example of cubic spline interpolation [24]

last), the cubic spline interpolation will not continue in the trend of the last points. In our case, since we do not use extrapolation (we only want to create a spline from the start point to the end point), we do not encounter this issue.

Lastly, using horizontal scaling combined with cubic spline interpolation is not recommended, as cubic spline interpolation is sensitive to horizontal scaling. When horizontal scaling is done with very few data points and only on independent variables, it causes unexpected oscillations and instabilities within the resulting spline. Within our implementation, we do not face this issue, because we scale both coordinates equally.

2.3.2 Road generation

The Deepjanus seed generator, included in the SBST21 code package [2], is an algorithm that can be used to create road representations. The Deepjanus seed generator creates a set of control nodes, which are the guide points for final road representation. Then the Deepjanus seed generator uses Catmull-Rom cubic splines[8] to create the full road representation using the control nodes. This road representation is then ready to be used by the simulator.

2.3.3 State-of-the-art Multi-objective genetic algorithm

When it comes to state-of-the-art multi-objective genetic algorithms, FITEST is a good example. They start with a set of randomly generated test cases as their initial population, where each solution is represented by a vector of values as mentioned in 2.3.1. After the simulation, they calculate the test objectives. With these test objectives, they aim to compute the distance to a failure (safety requirement violation). So, the objectives are specific to the features under test. In the original FITEST paper, they tested four self-driving features, Autonomous Cruise Control, Traffic Sign Recognition, Pedestrian Protection, and Automated Emergency Braking. Subsequently, test cases are evolved over successive generations using crossover and mutation, where the fittest parents and off-springs form the next generation.

[2] https://sbst21.github.io/

The proposed novel multi-objective genetic algorithm in this thesis is inspired by FITEST. However, in this thesis, we focus on the Lane Keeping Assistant as the main feature under test. Therefore our search objective revolves around measuring the distance to and Out of Bounds Episode (OBE), meaning a car driving outside the designated lane. This distance is inspired by the 'distance to failure' introduced by FITEST, but for a different feature and related to the potential violations of that feature.

On top of that, same as FITEST, we make use of a population and continuously improve the fitness of the population. However, because of the different solution representations, we are not able to make use of crossover, and thus we only use mutation. This is because our solution is represented by a list of points, and thus it is highly unlikely that if we perform crossover, we get two new valid roads.

2.4 Different sources for self-driving car test case generation

In order to generate test cases for self-driving cars, we need to establish a way of creating the roads. One way that is often used is generating random points. However, there have also been different sources used for creating these roads. One example of this is Gambi *et al.* , who used police reports of car crashes and recreated the environment of these crashes within a simulation [14]. Nguyen *et al.* proposed an automatic test case generator, which uses high-definition maps to automatically find quantifiably diverse and critical scenarios that can be used within simulation engines as test cases [29].

For our approach, we use a similar approach. We manually extract real-world roads using the MyMaps extension from Google and transform these real-world roads into usable scenarios within a simulation engine. This is different from the technique proposed by Nguyen *et al.* as they use a map in OpenDrive [12] format and turn it into abstract driving scenarios in which they can create test cases. Our approach uses the data from Google Maps and takes real-world roads using the MyMaps extension. We introduce this technique to create road scenarios that are as close to a one-on-one representation of the actual roads the self-driving cars will be driving on as possible.

Chapter 3

Approach

In this section, we describe how REWOSA generates test cases to find unsought results. In section 3.1 we explain how we extract and convert the roads from MyMaps into road points that we use to create a test case. In 3.2, we describe how we represent the roads used as test cases. Section 3.3 presents how these test cases are mutated in order to evolve our test cases. In section 3.4 we explain how we validate our generated test cases. Section 3.5 describes the test case selection procedure in which the algorithm selects the best candidates (*i.e.,* test cases) for mutation and generation of the new solutions. Finally, section 3.6 shows how REWOSA combines the previously explained components into an algorithm that generates scenarios to test for self-driving car systems.

3.1 Road extraction & conversion

We first need to gather a batch of real-world examples of roads. This is done with MyMaps, a Google service that provides a platform to use the Google Maps infrastructure to create custom maps by adding information to locations. We select roads based on if they are dangerous, because if a road is dangerous, it is more important for a self-driving car to make sure it can handle those types of roads. To decide which roads we choose, we use a website [1]where people submitted dangerous roads with data and articles about accidents and potentially even deaths that happened on that road. On top of that, these roads contain interesting sections that are hard to generate using a random road generator. An example of such a road is shown in figure 3.1. The road on the left is the road displayed in MyMaps. We use a `.kml` file containing the real-world coordinates of this road to then generate the road shown on the right. The image on the right is the end product of the conversion of the road coordinates into grid coordinates used in the simulation. Note that the start of the road is highlighted by the red dot. This road shows a variety between fast corners (*i.e.,* corners close to a straight line) and sharper corners. There are also sections that are harder to generate with randomness, such as the end part of the road, where we have a fairly constant radius corner into a chicane, highlighted by the green circle. We also included a couple of less dangerous or complex roads to provide a more balanced population, so approximately

[1] `https://www.dangerousroads.org/`

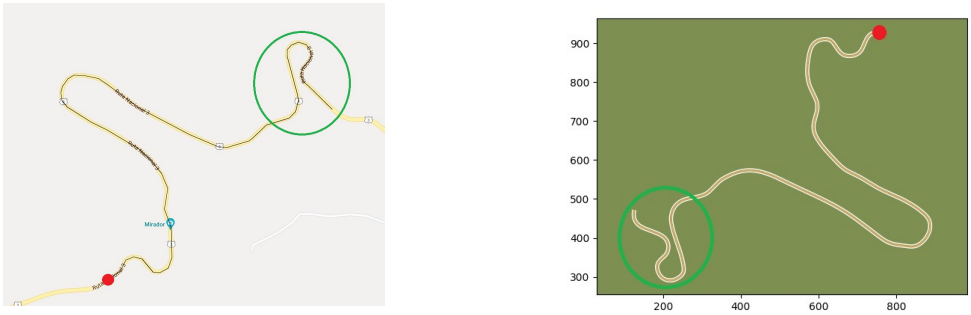

Figure 3.1: The image on the left shows the road extracted from Google Maps. The image on the right shows the same road, use within the simulation.

one-third of the population is filled with less dangerous roads. Once we have created a line in MyMaps, we can export that layer to kml and put all these kml files in a folder. We then use an extractor and converter in order to extract the coordinates from these files and convert these real-world coordinates into coordinates within the given grid. A depiction of the coordinate conversion algorithm is displayed in algorithm 1 and depicted in figure 3.2.

In order to convert real-world roads into usable roads within the simulation, we use the list of files used as input and loop over each file in this list, lines 1 to 23. These files contain a list of coordinates and much extra information. Therefore, in line 2, we start by extracting all the coordinates from the file and saving each coordinate in a list. Once this is done, we convert each coordinate into an integer instead of a decimal number in line 4. This is done as the actual world coordinates saved in the file are not usable without some adjustments. Once this is done, we save all the x and y coordinates in two separate lists, shown in line 7. In line 8, we then find the common prefix, and in lines 9 to 10, we remove the common prefix from the x coordinates list and y coordinates list and turn the remaining number into a decimal below ten. In line 12, we then take the minimum value from each coordinate list. Then, we subtract all x coordinates by the minimum x coordinate and the same for the y coordinate with the minimum y coordinate (lines 13 to 15). We then grab the maximum x and y coordinate, in line 16, to generate a value, done in line 17, with which we can multiply all our coordinates to fit them in the predetermined grid. We then multiply all our coordinates by this multiplication value (lines 18 to 21) and save the new coordinates in a list (line 22). Once all coordinates have been extracted and converted, we return the list of the new coordinates.

3.2 Road representation

The important part of the road representation is the area in which we can place the road. This is decided by the map size X given to the algorithm. This map size X then helps to create a grid of X by X that can be used to house the road.

This road is created from a list of coordinates. For the `REWOSA` generator, we use an x

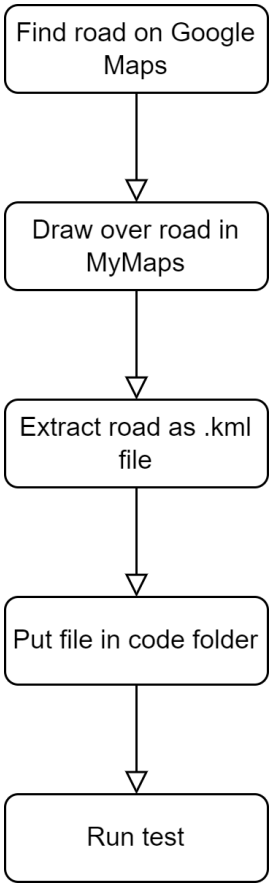

Figure 3.2: This flowchart depicts the flow of finding a road, extracting it and placing it in the code folder for it to be used.

and *y* value to translate to a point in the grid. When we have these coordinates, we use an interpolation technique to create an actual road. The interpolation technique uses the known data points (road points in our case) to estimate the unknown points (*i.e.,* parts in between the road points). When we combine these known points with the estimated points, we are able to create a road that can be used within the simulation engine.

For the `REWOSA` generator, we use normal spline [3] interpolation to create the road, which is explained in 2.3.1. We choose to use normal spline interpolation, as this gives a more accurate representation of how the road in the real world looks. On top of that, during preliminary research, test runs to compare spline interpolation to b-spline interpolation also showed that the spline interpolation was able to generate more failing test cases.

3.3 Mutation

The mutation of real-world test cases is done by adding a point in between two points of the test case, removing a random point, or adjusting a random point. The mutation is shown in

Algorithm 1 Coordinates conversion

Input: files = list of the .kml files with the data of the real-world roads.
Output: new_coords = list with converted coordinates, fitted to the grid.

```
1:  for file in files do
2:      c ⟵ parse_coordinates(file)
3:      for p in c do
4:          p ⟵ transform_coordinates(p)
5:          store_coordinates(p)
6:      end for
7:      x_list, y_list ⟵ store_x_y_separate(c)
8:      pre ⟵ common_prefix(new_c)
9:      for i in range(len(x_list)) do
10:         x_list[i], y_list[i] ⟵ remove_pre(x_list[i], y_list[i])
11:     end for
12:     x, y ⟵ get_smallest(x_list, y_list)
13:     for i in range(len(x_list)) do
14:         x_list[i], y_list[i] ⟵ remove_int(x_list[i], y_list[i])
15:     end for
16:     max ⟵ get_max(x_list, y_list)
17:     mul ⟵ (grid_size − 20)/max_value
18:     for i in range(len(x_list)) do
19:         list_x[i] ⟵ list_x[i] · mul
20:         list_y[i] ⟵ list_y[i] · mul
21:     end for
22:     new_coords ⟵ store_final_coords(x_list, y_list)
23: end for
```

Algorithm 2. We start by randomly selecting if we want to add, remove, or adjust a point in line 1. For the addition of a point (lines 3 to 16), the place where we add this point is decided based on the two points that are the furthest away from each other, calculated in lines 4 to 11. In line 12, we calculate the line perpendicular to the line between the two consecutive points furthest from each other. We then grab the middle point of the line between the two points furthest from each other (line 13). Next, a new point will be placed on the perpendicular line through the middle point of the line between the two points furthest from each other, done in line 14. When removing a point (lines 17 to 20), we randomly select a point, line 18, and then remove said point, done in line 19. When adjusting a random point (lines 21 to 28), we first randomly select which point we will adjust in line 22. Then, we generate a random adjustment for both the x and y coordinate, in line 23, between negative five and five. We then check if the point with this adjustment is still within the map and not exactly another point. If this is the case, we re-generate the adjustments and do the same check until it is a correct point, done in the loop of lines 24 to 26. We then update this point with the adjustments and return the list of points in line 27.

Figures 3.3, 3.4 and 3.5 demonstrates examples of this mutation operator. For the ad-

Algorithm 2 Real-world Mutation

Input: points = list of the road points.
range = maximum value to be added or subtracted from a coordinate.
Output: points = updated list of the road points.

```
1:  action ⟵ select('add', 'remove', 'adjust')
2:  max_dist ⟵ 0
3:  if action is 'add' then
4:      for i in range(points) do
5:          dist ⟵ get_distance(points[i], points[i+1])
6:          if dist > max_dist then
7:              max_dist ⟵ dist
8:              point_a, point_b ⟵ points[i], points[i+1]
9:              index ⟵ i
10:         end if
11:     end for
12:     perp ⟵ get_perpendicular(point_a, point_b)
13:     mid ⟵ get_mid_point(point_a, point_b)
14:     new_point ⟵ get_new_point(perp, mid)
15:     points ⟵ insert(index + 1, new_point)
16: end if
17: if action is 'remove' then
18:     i ⟵ randint(len(points))
19:     points ⟵ remove(points, i)
20: end if
21: if action is 'adjust' then
22:     i ⟵ randint(len(points))
23:     addition ⟵ get_rand_add(range)
24:     while points[i] + addition NOT in grid do
25:         addition ⟵ get_rand_add(range)
26:     end while
27:     points[i] ⟵ points[i] + addition
28: end if
```

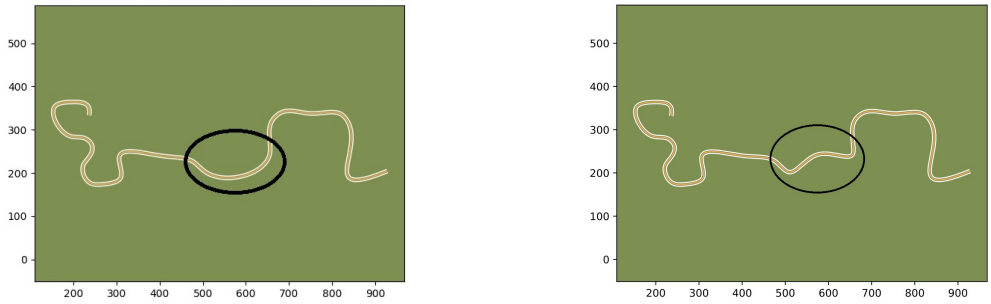

Figure 3.3: The image on the left shows the road before adding a new point. The image on the right shows the same road after adding a new point.

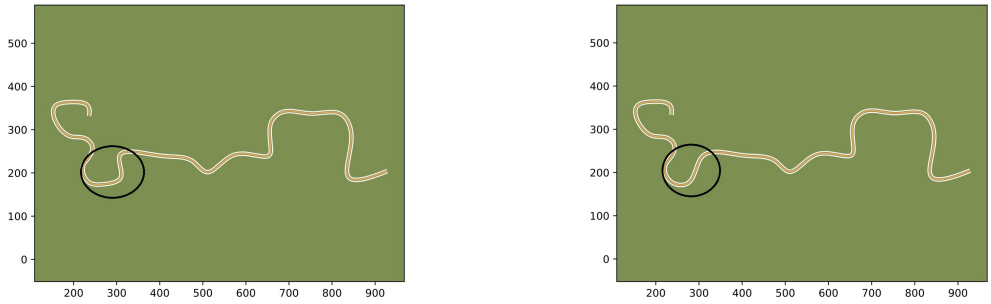

Figure 3.4: The image on the left shows the road before adjusting a point. The image on the right shows the same road after adjusting a point.

dition of a point, we look at figure 3.3, where the point that was added from the first to the second road is highlighted with black circles. Figure 3.4 shows the adjusting of a point. In this figure, the adjusted point from the first to the second road is highlighted. Finally, for the removal of a point, look at figure 3.5 in which the black circles highlight the removal of a point from the first road.

3.4 Validation

In order to make sure that the provided road is actually a road that is usable in the simulator and could be considered as an actual road, we perform multiple validity checks.

For the first check, we examine the radius between consecutive points to find the sharpest corner within the generated road. If this radius turns out to be smaller than 47°, the corner is considered too sharp. If the road contains a corner that is too sharp, we will cut off the road just before this corner and simulate the resulting road. An example of a too sharp corner is shown in figure 3.6, highlighted by the number 1.

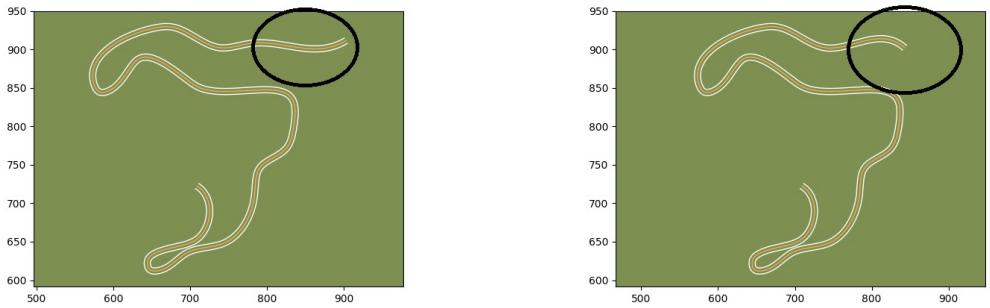

Figure 3.5: The image on the left shows the road before a point was removed. The image on the right shows the same road after a point was removed.

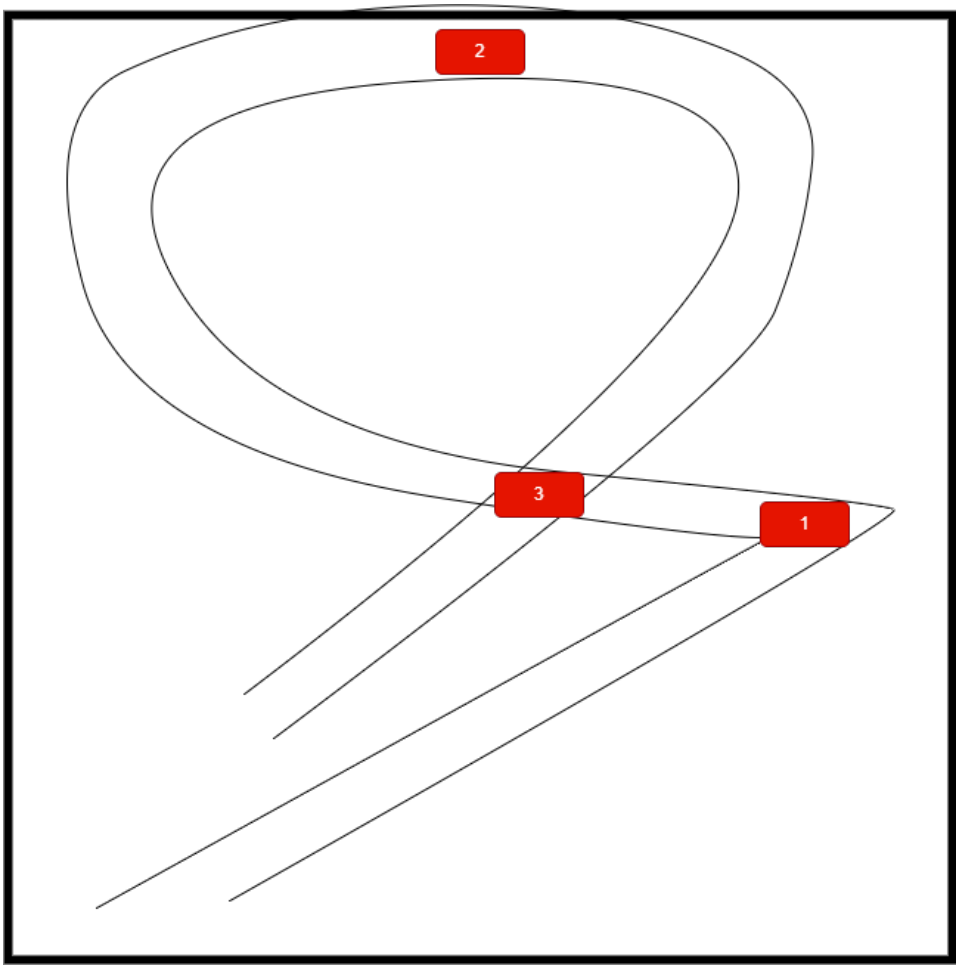

Figure 3.6: Example of validation violations, with 1 showing a corner that is too sharp; 2 showing that the road is outside of the grid; and 3 showing that the road is self-intersecting.

We also check whether or not the given road data is of the correct type to ensure that we can simulate the test correctly. That means whether the object we will give to the simulation engine is a road and not something else.

When we create a road, it is important that we generate a road that actually is a road. In order to do this, we ensure that the road contains at least 2 points. On this topic, we also make sure that the road is not too short. Hence, we check if the road is longer than 20 meters. We also check if the road does not consist of too many road points, resulting in a large simulation time. In this case, we check if the road consists of less than 500 points.

Because we are working with a grid in which we place the road, it is also essential to check if the road is actually within the grid. To do this, we have two checks: (i) check if the road is within the grid, and (ii) check if the road intersects with the grid boundary. An example of the road being outside of the given map is shown in figure 3.6. Here we can see the black box, which is the map, and at the number 2, we can see that the road crosses outside the map boundaries.

Due to the road being in 2D, it is not wanted for the road to intersect with itself. This check includes two parts of the roads crossing each other as well as two parts of the road overlapping each other. When we look at figure 3.6 we can see at number 3 that the road crosses over itself.

3.5 Search Objectives

Our search algorithm aims to find the test cases that exceed the given out-of-bounds tolerance. The aim is to generate test cases in which the car goes out of bounds for a certain percentage. Therefore, we only want to keep the non-failing test cases that are very close to failing. To decide if a test case is close to failing, we look at the worst state for every three objectives we set. In our case, the objectives are (i) the out-of-bounds (oob) area, (ii) velocity, and (iii) the steering angle of the car. For each of these objectives, we select the ”best” state, which means the state with the highest oob area, velocity, and steering angle, respectively.

The following formula shows the oob area of the ”best” state in a single test case:

$$OOB\_area(f) = \max_{i \in 1,\ldots,n} (area\_car(f_i) - overlap\_area(f_i)) \tag{3.1}$$

where i goes from 1 to n and represents the $i-th$ frame of the simulation, meaning the equation above will find the frame in which the car is outside of its lane with the largest area; $area\_car$ refers to the total area of the car, viewed from a 2D perspective (top-down view); and $overlap\_area$ refers to the area of the car that overlaps with the lane it is supposed to drive in.

The velocity and steering area are represented by a single value and can be formulated to the following formulas:

$$velocity(f) = \max_{i \in i,\ldots,n} (velocity(f_i)) \tag{3.2}$$

$$steering\_angle = \max_{i \in i,\ldots,n} (steering\_angle(f_i)) \tag{3.3}$$

where i goes from 1 to n and represents the $i-th$ frame of the simulation, meaning the formulas will find the frame with the highest velocity and steering angle respectively.

Now that we have a representation of the test case, we compare the new test case with the tests that are already saved within the non-dominated list. This non-dominated list represents the Pareto front of the passed test cases. Thus, when we have a new test case, we compare it with all other test cases in the current Pareto front and update the Pareto front accordingly. The comparison works by comparing the highest oob area, highest velocity, and largest steering angle between the two test cases. When one test case has an oob area, velocity, and steering angle greater or equal to the other test case, then that test case dominates the other test case. If the new test case gets dominated by any of the test cases in the Pareto front, it will never be able to dominate any other test cases in the Pareto front. So, we can stop the comparison. In the case that the new test case does dominate a test case already in the Pareto front, the test case that was in the Pareto front will get removed from the list. In case the new test case does not get dominated by any of the test cases already in the Pareto front, the new test case will be added to the list, representing the Pareto front.

3.6 REWOSA

For REWOSA we have to perform some minor manual preparation work, which is explained in 3.1. After the initial preparation, we have a list of all the real-world roads. We already explained the road representation in section 3.2. We randomly select one from the list and interpolate it using spline interpolation, which is the same as used for and explained in the background in 2.3.1. We then check this road on validity, explained in 3.4. If it is valid, we execute the test case. Otherwise, we generate a new test case. Once the test is executed, we get the test outcome returned. Given the test outcome, we can have three situations:

- The test case returns Failed. In this case we store this failing test case in a list with all other failing test cases.
- The test case returns Passed. If the test passes, we compare the passing test case with all previous non-dominated passing test cases using the search objectives explained in 3.5.
- The test case returns Error. This occurs when a error happened during the test run. In this case we move on to the next test case.

After the test case is stored or removed, we generate a new test case. How we select whether we pick a new real-world scenario or mutate an already executed test case is as follows. We slowly inject real-world scenarios, meaning we will start with taking five random roads from the list of extracted coordinates. Next, we mutate five test cases and then simulate one new real-world scenario in a loop. If no more roads are left in the list of extracted coordinates, we randomly grab a test from the non-dominated passing test cases and mutate

Algorithm 3 Main code loop REWOSA

Input: batch_id = id of folder that stores each test case from a single run.
result_folder = folder that keeps the results from a single test run.
map_size = size of the grid used for the roads.
time_budget = time budget allocated to the test run.

```
1:  non_dominated_list ⟵ []
2:  oob_test ⟵ []
3:  radii_list ⟵ []
4:  start_time ⟵ time()
5:  coordinates_list ⟵ extract_coordinates(map_size)
6:  time_budget ⟵ time_budget - (time() - start_time)
7:  while time_budget > 0 do
8:      start_time ⟵ time()
9:      road_points, start ⟵ get_points(coordinates_list)
10:     if test is unique then
11:         the_test ⟵ create_road_test(road_points)
12:         test_outcome, execution_data, radii ⟵
              execute_test(the_test, batch_id, start)
13:         if test_outcome == 'FAIL' then
14:             oob_tests.append(road_points[:])
15:             radii_list.append(radii[:])
16:         end if
17:         if test_outcome == 'PASS' then
18:             scores ⟵ get_scores(execution_data)
19:             non_dominated_list ⟵ update_non_dom(road_points[:], scores)
20:         end if
21:     end if
22:     current_time ⟵ time()
23:     elapsed_time ⟵ current_time - start_time
24:     time_budget ⟵ time_budget - elapsed_time
25: end while
26: scores ⟵ calculate_jaccard_score(radii_list)
27: store_jaccard_score(scores, result_folder)
```

that test case. How a test case is mutated is described in section 3.3. We then calculate the Jaccard similarity score when the time budget is finished to measure the similarity of the failing test cases. The full code loop of `REWOSA` is shown in algorithm 3.

In this algorithm, the singular form of radii means the angle created between a point and the second and fourth point after it. We then have the `start_time`, `coordinates_list`, which has the list of real-world road scenarios. Next, we update the time budget with the time used to extract the `coordinates_list`. We will start the main loop (lines 7 to 25) until the time budget is finished. Within the loop, we start by setting the start time, which is used to keep track of the consumed time, so we can update the time budget. Then, we grab the road points, line 9, described in the paragraph above, and state if these points are generated from a real-world scenario or if they are mutated. If the test is unique, meaning it is not the same as an already failed or passed test case, we create the road test, line 11, which is the road that the simulator can actually use. In line 12, we execute the test, giving an outcome (pass/fail/error), the execution data, and the radii that make up the road in return. In case of receiving a fail outcome (lines 13 to 16) we add the road points of that test case to the list of failing test cases and also add the `radii` of that test case to the `radii_list`. In case of receiving a pass outcome (lines 17 to 20), we generate a score and then use this score in order to update the non-dominance list. At the end of the loop (lines 22 to 24), we update the time budget by using the current time and the start time. Once the time budget is over, we use the `radii_list` to calculate the similarity between all the failing test cases using the Jaccard similarity and store the results.

Chapter 4

Empirical Study

4.1 Baseline

In order to assess the performance of our novel approach, we need to compare it to the baseline. The baseline is similar to REWOSA, but there are some differences in (i) how the roads are represented, (ii) how we handle the mutation, and (iii) in the main code loop, which will be explained in the following sections. In general, we design this baseline according to the best practices in automated test generation.

4.1.1 Road representation

For our novel approach REWOSA the roads are represented by an x and y coordinate. For the baseline, we use x and y values as well, but it also adds a z and *width* value. Here, the z value represents the vertical coordinate of the road. As the road is all on the same plane, this z value will be the same for each point of the road. The *width* value states the width of the road. As we do not want any narrowing or widening, this *width* value will also be the exact same for each road point. So even though this version has these extra z and *width* values, both of these values are constant for each of the coordinates used in the baseline.

The representation of the roads differs because for the baseline we make use of the Deepjanus seed generator to create the roads, resulting in each point consisting of an x, y and z coordinate and a width. Whereas for REWOSA the coordinates come from the real-world coordinates stored in the kml files, which only contain and x and y coordinate.

4.1.2 Mutation

The mutation is applied to a solution (here road) randomly selected from the population. For the mutation, there are three different possibilities, which all have an equal chance of happening: adding points, adjusting points, or removing points. When removing points, we first randomly select a point and then remove that point, as well as the five points before and after that point. An example of this can be seen in figure 4.1, where highlighted with the black circle we can see points being removed from the first to the second road. When we add points, we let the road generation algorithm generate a new node to extend the

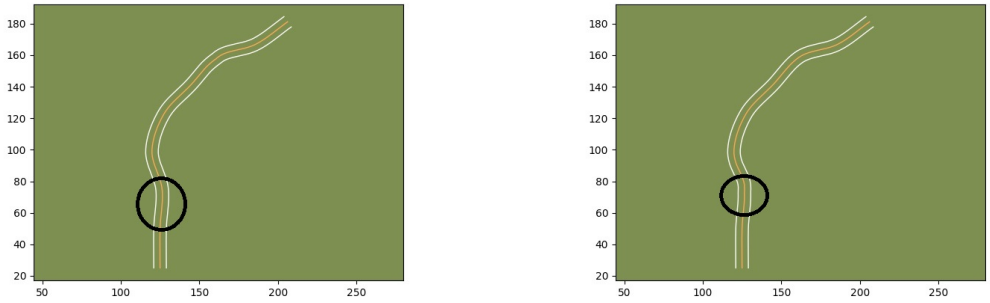

Figure 4.1: The image on the left shows the road before a point was removed. The image on the right shows the same road after a point was removed.

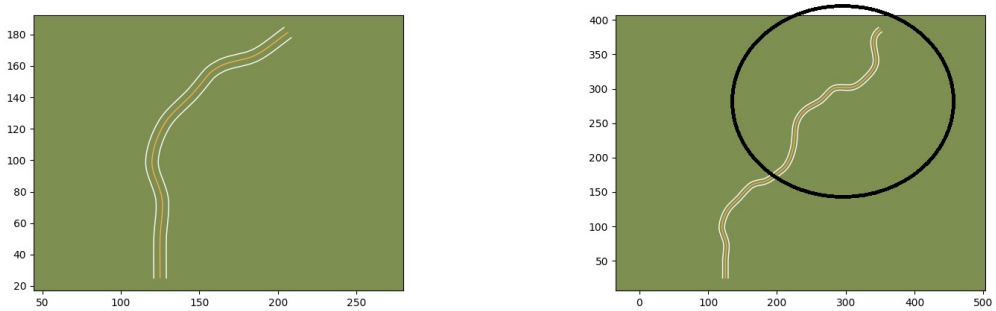

Figure 4.2: The image on the left shows the road before adding a new point. The image on the right shows the same road after adding a new point.

initial road, up to eleven nodes. We then check if this new point is within the bounds of the map. If not, we remove the new point and adjust the points instead and then stop the mutation. We can see the addition of nodes in figure 4.2, where it is highlighted with the black circle. When adjusting points, we randomly select a point and adjust both coordinates of this point between -2 and 2 compared to the initial value. We also adjust the five points before and after this point with the same adjustment, but linearly decreasing the further, we get away from the initially adjusted point. This is done to avoid random sharp turns due to the single-point adjustment. This can be seen in figure 4.3, highlighted by the black circle.

The reason for the different mutation strategy boils down to the different number of points. Because there is a different number of points between the baseline and `REWOSA`, we also add, adjust and remove a different number of points.

4.1.3 Main loop

For the baseline we start of by generating a initial population using the Deepjanus seed generator, explained in 2.3.2. When we have the road points and turn it into an actual road,

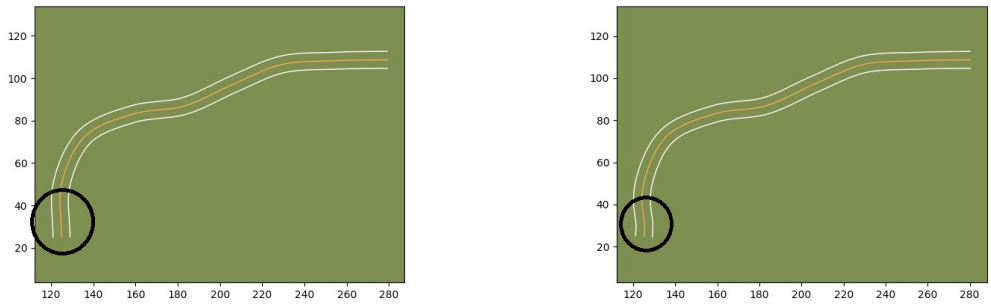

Figure 4.3: The image on the left shows the road before adjusting a point. The image on the right shows the same road after adjusting a point.

Algorithm 4 Main code loop baseline

Input: batch_id = id of folder that stores each test case from a single run.
result_folder = folder that keeps the results from a single test run.
time_budget = time budget allocated to the test run.

```
1:  non_dominated_list ⟵ []
2:  oob_test ⟵ []
3:  radii_list ⟵ []
4:  while time_budget > 0 do
5:      start_time ⟵ time()
6:      road_points, start ⟵ get_points()
7:      if test is unique then
8:          the_test ⟵ create_road_test(road_points)
9:          test_outcome, execution_data, radii ⟵
              execute_test(the_test, batch_id, start)
10:         if test_outcome == 'FAIL' then
11:             oob_tests.append(road_points[:])
12:             radii_list.append(radii[:])
13:         end if
14:         if test_outcome == 'PASS' then
15:             scores ⟵ get_scores(execution_data)
16:             non_dominated_list ⟵ update_non_dom(road_points[:], scores)
17:         end if
18:     end if
19:     current_time ⟵ time()
20:     elapsed_time ⟵ current_time - start_time
21:     time_budget ⟵ time_budget - elapsed_time
22: end while
23: scores ⟵ calculate_jaccard_score(radii_list)
24: store_jaccard_score(scores, result_folder)
```

we make use of spline interpolation (2.3.1). Then we check if the created road is valid(3.4), and if so we execute the test case. After executing the test case, we get given three outcomes, Failed, Passed or Error. In the case that the result is Failed, it will be added to a list of all the failed test cases. When it Passes, we compare the fitness of the test case with all previously non-dominated passing test cases. In case of an error, the test case will be ignored and we move on to the next test case. Once we have three or more passing test cases in the Pareto front, we randomly select a test case from this list and mutate it, according to the algorithm explained in 4.1.2. Then when the time budget is up, we calculate the Jaccard similarity score between the failing tests, to see how diverse the failing roads are.

The baseline code is shown in algorithm 4, where we start of by initializing some variables in lines 1-3. Then the baseline starts of by generating the coordinates, in line 6, as long as we still have time budget left. In order to generate the coordinates it uses the Deepjanus seed generator. It then interpolates these coordinates using the spline interpolation technique, in line 8. When the road is created, we check the validity of the road. If the road, and thus the test case, is valid, we execute the test, seen in line 9. If the road turns out to be invalid, we generate a new road. Once the test is executed we get the test outcome returned. Given the test outcome, we can have three situations:

- The test case returns Failed. In this case we store this failing test case in a list with all other failing test cases in lines 10 to 13.
- The test case returns Passed. If the test passes, we compare the passing test case with all previous non-dominated passing test cases, seen in lines 14 to 17.
- The test case returns Error. This occurs when an error happened during the test run. In this case we move on to the next test case.

After the test case is stored or removed, we generate a new test case. If the list of non-dominated passing test cases has less than three test cases, we continue to generate random test cases. If the list of non-dominated passing test cases has three or more test cases, we randomly (uniform) select a test case from the non-dominated passing test case list that we can mutate. In lines 23 and 24, when the time budget is fully used up, we calculate the Jaccard similarity score between all failing test cases to see how similar they are. We do this because it could be the case that multiple failing test cases are mutations of the same test case, resulting in very similar road structures.

4.2 Benchmark

To assess `REWOSA` against baseline in terms of their ability to generate failing test cases, we compare the tests generated by each of these tools. As we use the SBST21 competition tool as the base for our generators, we use similar values from the SBST21 setup for our setup. Using similar values as SBST21 setup leads to easier comparison between the tools introduced in this thesis and the tools that are evaluated in the SBST21 competition.

In order to evaluate the approaches in this study we use the BeamNG.tech driving simulation engine, and the built-in BeamNG.AI driving agent as research subject. The main rea-

son for using this simulator and research subject is to provide an easier comparison between the tools used in our research and the tools from the SBST21 tool competition. Moreover, it has also been used in other previous research [14, 15, 27] to evaluate test input generators and train vision-based steering predictors and it provides a realistic simulation of the roads. There is also no manual training required, which lowers the threats to validity. Lastly, we can use different parameter values, such as maximum velocity, to adjust the driving style.

The aim of each of these approaches is to generate as many diverse failing test cases within the given time budget. This is achieved by generating a road that is complex enough for the car to partially drive outside of its lane. To decide whether a self-driving car is outside of its own lane, we check the given tolerance. As an example, if the tolerance is 0.75 it means that more than three quarters of the car needs to be outside of its lane in order to consider that test case as failing. Besides, for measuring the diversity of the failing test cases we use the Jaccard similarity score as mentioned in section 3.

Table 4.1: Experimental Setup

Name	**Map Size** (m^2)	**Max Speed** (Km/h)	**Budget** (h)	**Tolerance** (%)
Default	950×950	-	2	0.95
SBST21	950×950	70	2	0.85

We evaluate the different tools on our research subject using two different experimental setups, Default and SBST21 (see Table 4.1). With the Default setup we have set a time budget of two hours, we use a tolerance of 0.95, and we do not set a maximum speed for the car, in order to provoke a more careless style of driving. With the SBST21 setup we have set the time budget to two hours, with a tolerance of 0.85 and a speed limit of 70 Km/h to ensure a more cautious driving style.

The considerable difference between our experimental setup and the setup used in the SBST21 tool competition is the map size. For our setup, we use a map size of 950×950, instead of the 200×200 map size used in the tool competition. The reason why this is done is due to the use of the real world roads. According to our performed pre-analysis in this thesis, a map size smaller than 900×900 leads to 90% or more invalid tests. Thus, in order to provide for a solid comparison between the tools, we pick a map size that is able to deliver a fair amount of valid test cases for both tools.

We gather the final results by running the tests locally on a 5-year-old windows machine. This machine contains a Intel Core i5-6300HQ CPU, a Intel HD graphics 530 graphics card and 7.8 GB usable RAM.

4.3 Research Questions

The main aim of this research is to provide new insights in how effective the use of real-world roads is, in order to reveal malfunctions within the self-driving software. As described before these real-world roads are hand selected in order to provide a balance between complex or dangerous roads and more straightforward roads. As there have not been

any research on the use of real-world roads before, we want to assess how well they would perform compared to a well known and widely used random generated roads. In order to assess this, we formulate the following research questions:

RQ1 *How complex are real-world road scenarios compared to randomly generated roads?*

To answer this question we focus purely on the structure of the two types of roads, real-world roads and the randomly initialized roads. With this question the focus is on the complexity of the roads, and what the impact of the complexity is.

RQ2 *How well does* `REWOSA` *perform compared to the baseline?*

To answer this question, we compare our newly suggested road generator `REWOSA` against the baseline in terms of generating test cases that effectively make the car drive outside of the designated lane. To go further into this question, we look into the diversity between the failing test cases, because creating a lot of test cases that fail is not necessarily helpful, as these failing test cases may all be similar.

RQ3 *What is the overhead of the real-world road seeding?*

We answer this question by looking at the overhead required for the real-world seeding approach `REWOSA`. To do this we use the following sub-questions:

$RQ_{3.1}$: How large is the preparation work required for `REWOSA`*?*
$RQ_{3.2}$: How does the creation and validation time of the road compare between the baseline and `REWOSA`*?*

4.4 Study Design

For the first research question, regarding the complexity of the roads, we aim to extract information by using a tool created by Birchler called sdc-scissor [1]. With the help of this tool, we can extract features from a road after the test runs have been executed. The tool collects road points from the test results and uses that to split the road up into segments. These segments are then used to calculate various features from the road. For **RQ1** we use the number of left turns, right turns, and straights, as well as the total distance of each road.

To provide an answer to **RQ2**, the results are generated by both the baseline and the `REWOSA`, with both the Default and SBST21 setup, with twenty runs for each generator-setup combination. In order to compare the tools included in this experiment, we collect and analyze (i) the number of valid test cases compared to the total number of generated test cases, (ii) the number of failing test cases, and (iii) the number of unmutated failing test cases.

[1] `https://github.com/ChristianBirchler/sdc-scissor`

We also highlight the number of unmutated failing test cases, mainly for the generator that uses real-world roads. This is because the real-world roads used are hand selected and gathered while using Google Maps. If most of the failing test cases from these generators come from the unmutated test cases, then that means that the real-world roads were very well selected, but the mutation part of the generator is not good. On the other hand, to give a better insight into how well the genetic algorithm part of the generators works, is also why we distinguish between test cases that fail on first creation and test cases that fail after mutation. In order to draw accurate conclusions from these results, we also provide statistical test results, which show the significance of our results.

The statistical tests that are used for this are the Vargha-Delaney A measure [40] and the Wilcoxon rank sum [7]. The Vargha-Delaney A measure calculates the effect size between two populations. The value will be between 0 and 1, where a value of 0.5 means that both populations perform equally. When the A measure value is below 0.5, the first population performs worse than the second population; likewise, when the A measure value is above 0.5, the second population performs worse than the first population. The Wilcoxon rank sum calculates a P-value, which describes if the difference between the two given sets is statistically significant. A widely used P-value from which point on-wards we would classify a difference as statistically significant is 0.05 and below [5]. For these statistical tests, we used the number of failing test cases per run. Regarding the effect size measure, when comparing the different generator-setup combinations in the results section, the better performing generator is highlighted in dark green.

We also provide similarity calculations between failing test cases from the same test run. This helps to highlight how diverse the failing test cases generated in the same test run are.

To measure the overhead that comes with using real-world roads (**RQ3** and $RQ_{3.1}$), we provide an overview of the preparation work needed to get all the roads. On top of that, we will also look into the time required for the process to convert the road coordinates into a road used by the simulation engine to provide an answer to $RQ_{3.2}$.

4.5 Parameter Setting

For the population size of the baseline, we start by randomly initializing roads until there are three non-dominated roads. Once this is done, we continuously mutate roads randomly selected from these non-dominated roads. If the number of roads in the non-dominated list is lower than three at any point, we randomly initialize a new road until there are three non-dominated roads again. For the `REWOSA` we start with a population of thirty real-world roads. We start by randomly picking five of these roads to execute. We then randomly pick a non-dominated road and mutate that road, this process happens five times, after which we grab a new road from the initial population to execute. If the non-dominated road list is empty, and we still have unused roads in the initial population, we will randomly select a road from the initial population. On the other hand, once the roads from the initial population are all used, we will continue to mutate roads from the non-dominated road list. The actual population size of the non-dominated road list is not limited, so it can be ever-growing.

As mentioned, when mutating roads, for both the baseline and the REWOSA generator, there is a complete even chance to either add a point, remove a point or adjust a point. However, for the baseline, in order to be allowed to remove a point, the selected road must at least exist of more than three road points. In order to be allowed to add a point, the road must have less than ten road points. The lower margin is set as a road of two points is just a straight line, which is not complex enough to be used as a test. The upper margin is set as preliminary research has shown that roads containing more than ten points start mainly generating invalid roads. For the REWOSA there are no extra restrictions set for adding, removing, or adjusting road points, as these roads contain a large number of road points, meaning adding or removing a point will not have a large enough impact to cause straight roads or a large number of invalid roads.

To calculate the non-dominance scores, we count the normalized OOB area once and the normalized velocity and steering angle twice. These values were selected based on preliminary research, which showed a slightly better, but almost insignificant, performance than counting all three objectives as even.

The algorithm's timeout is set to two hours, which is more in line with the SBST setups. The main change we made compared to SBST is to use a two-hour time budget for both of the setups. This budget provides a better comparison between the baseline and the REWOSA as there is less difference in variables of the experimental setups.

Chapter 5

Results

5.1 RQ1: Real-world roads vs. random generated roads

For the comparison between real-world and randomly generated roads, we look at the type of segments (left turn, right turn, straight) used to create the road, as well as the total road distance. An overview of the type of segments for each combination of generator and setup is shown in figure 5.1.

The figures for REWOSA show that it creates roads with an average of more then ten left and ten right turns and five straight segments. In addition, we see that the mean is close to ten for the left turns, above the ten for right turns and around 5 for the straights. With the SBST21 setup, REWOSA can generate a 9.5% more segments than with the default setup. We can also see that almost all the averages are above the median, which is also indicated by the fact that most box-plots have a larger Q3 and Q4 compared to Q1 and Q2, meaning that the roads with a higher number of segments have a vast amount more, whereas the roads with a lower number of segments have a more similar amount.

The figures for the baseline show us that the roads created by the baseline have an average and mean of around five left runs, five right turns and five straights. Similar to REWOSA, the baseline seems to create about 7.4% more segments with the SBST21 setup compared to the default setup. We also notice here that the average lays above the mean, which we also identify when looking at the box-plots from REWOSA. Something interesting that we do recognize is that the baseline generates a good chunk of outliers.

If we compare REWOSA with the baseline, we identify that the baseline produces half the amount of left and right turns while it creates about the same number of straights. As noted before, the baseline produces some outliers, whereas REWOSA produced only one outlier. This shows that REWOSA is more stable with the production of the number of segments than the baseline.

In figure 5.2, we see box-plots that display the distance of the roads created by the different generator and setup combinations. The baseline shows that it creates most roads with a distance of approximately 250 meters when it comes to the default setup and creates roads of about 250-500 meters with the SBST21 setup. We notice a considerable number of outliers with the baseline and default setup, which is also highlighted by the fact that the

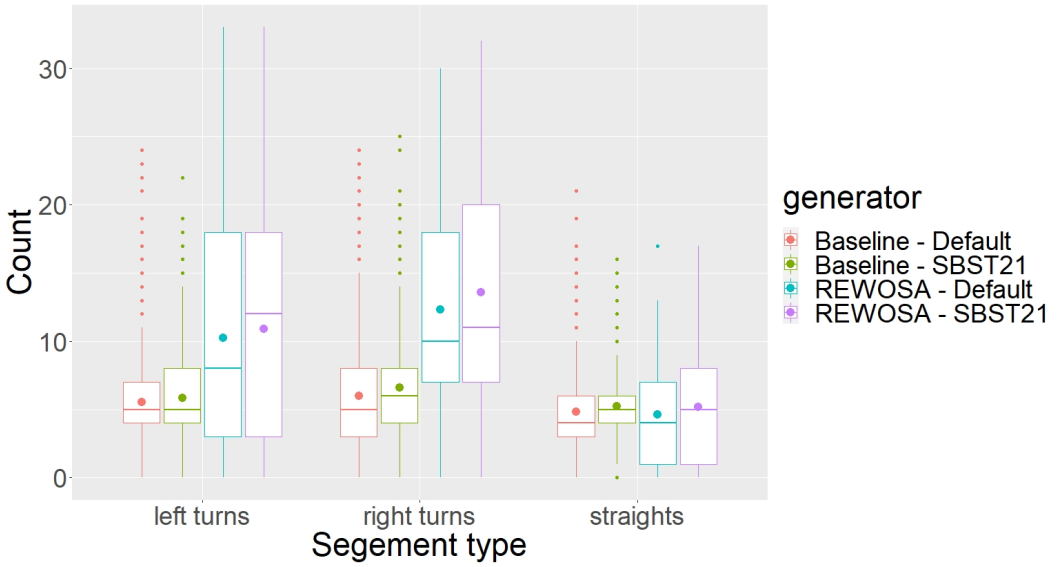

Figure 5.1: Overview of the different types of segments within the roads created by the different combinations of generator and setup. These figures display box-plots in which the outliers are shown with small dots and the average value is displayed by a big dot.

average is above Q4, which says that the number of outliers above Q4 will also be higher than the number of outliers below Q1.

With `REWOSA` the roads are approximately 1500 meters in size, with a margin of about 500 meters to either side. The average values lay close to the median, meaning that the roads, as well as the couple outliers, are well balanced in size.

When we compare the baseline with `REWOSA`, we notice a big difference in size. The roads from the baseline are about 3 to 6 times as short as the roads from `REWOSA`. Due to the box-plot shapes, we can also identify that the baseline creates a lot more roads with very similar sizes, whereas `REWOSA` has a better size variation.

What we have seen from these results is that the roads generated by `REWOSA` generate more segments than the baseline. This goes in line with the distance of the roads generated by `REWOSA` and the baseline, as the roads from `REWOSA` are 3 to 6 times bigger than the roads from the baseline.

RQ_1: How complex are real-world road scenarios compared to randomly generated roads?

In general, the real-world roads are longer and thus also contain more left and right turns. Due to the increase in both size and the turns, which are more complex than straights, real-world roads' complexity is higher than randomly generated roads.

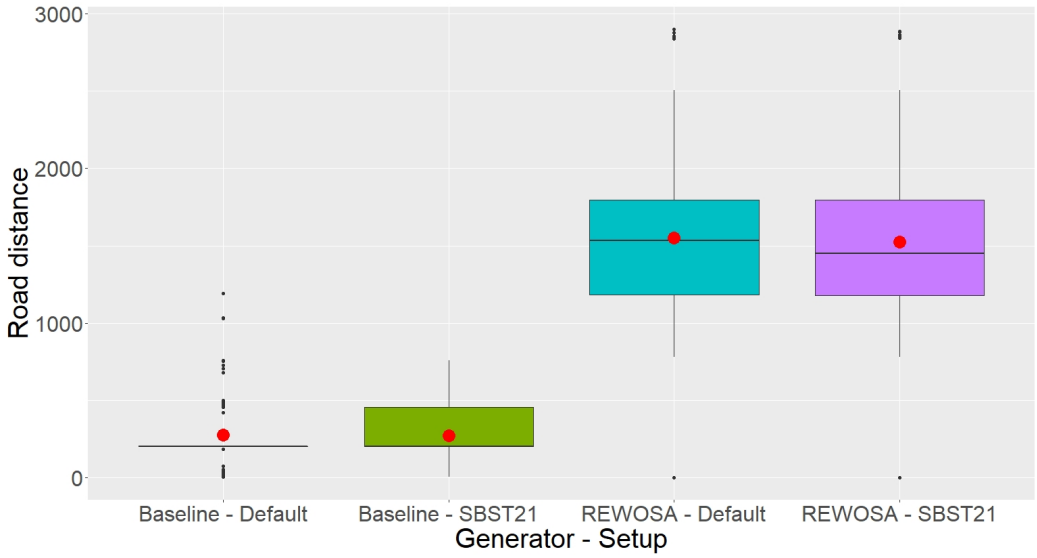

Figure 5.2: Overview of the length of the roads created by the different generator and setup combinations.

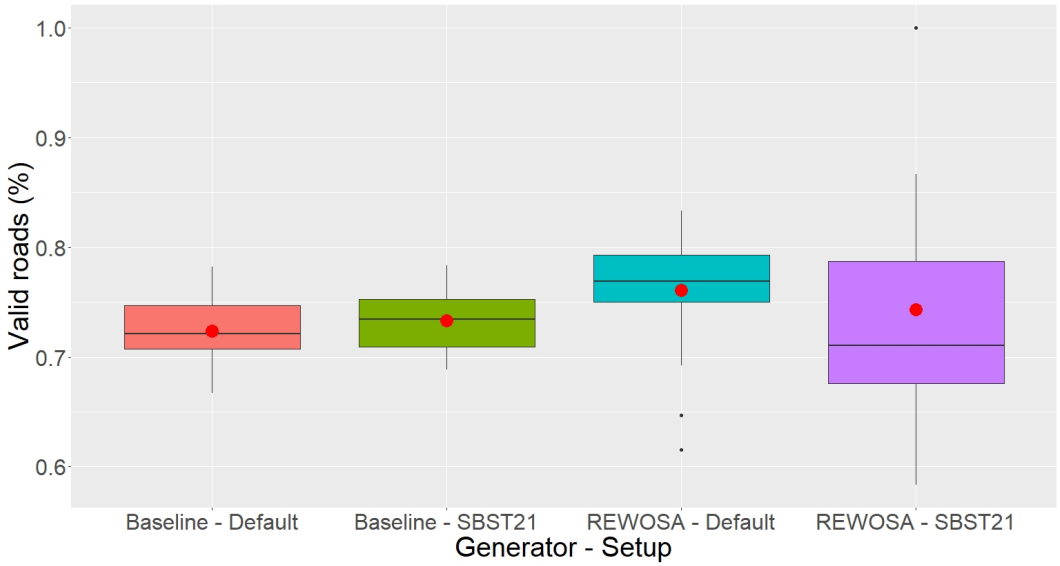

Figure 5.3: Overview of the number of valid test cases (roads) created over the total number of test cases (roads) by the different generator and setup combinations.

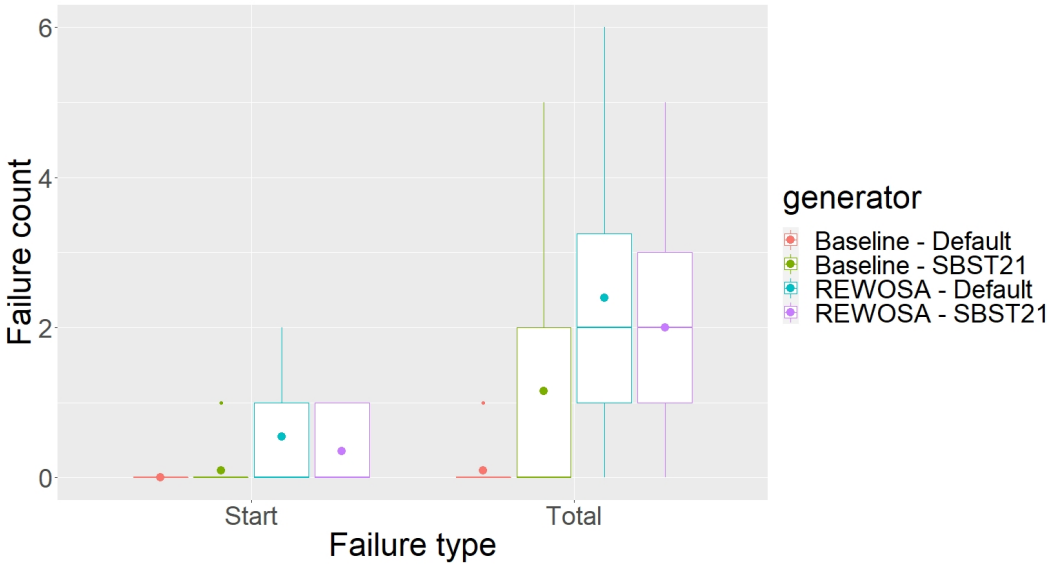

Figure 5.4: Overview of the number of failing test cases (roads) created by the different generator and setup combinations.

5.2 RQ2: Fault detection REWOSA vs. baseline

Figure 5.3 gives an overview of the number of valid roads over the total amount of roads generated. It shows that the baseline and REWOSA perform very similarly with a 70 to 80% valid roads rate. Only REWOSA with the SBST21 setup has a wider range of valid roads with approximately 60 to 85%. Additionally, on average, REWOSA achieves a higher number of valid test cases compared to baseline in both setups.

Figure 5.4 demonstrates the number of failed test cases, which indicates a larger number of failing test cases for REWOSA compared to the baseline. We see that the average for the baseline reaches 0 and just above 1, whereas the average for REWOSA is above 2.

In order to see how big the influence of the initial population of real-world roads is, we also look at the number of failed tests compared to how many of these are from initial creation and not mutated. Here we can identify that REWOSA does have more test cases failing at the start compared to the baseline. This shows that the initial population of the real-world roads has an impact on the number of failing test cases.

In order to provide any significance to the results displayed in figures 5.3 and 5.4, we also provide statistical test results. In table 5.1, we display the statistical tests based on the number of failed test cases.

In order to show that our results are significant, we have also performed some statistical tests, seen in table 5.1. If we look at the table, we can see that the effect size, when comparing the baseline to REWOSA, for both setups it shows that REWOSA outperforms the baseline. We can also see that the p-value from the Wilcoxon rank sum is also below the set threshold of 0.05, showing that they are highly unlikely from the same population. For the baseline, we can see that the baseline performs better with the SBST21 setup compared

Table 5.1: This table shows results of statistical tests performed on sets failures generated by two different generator(setup) combinations using the Vargha-Delaney A measure for the effect size and the Wilcoxon rank sum for the p-value.

Generator(setup) comparison	Effect size measure	P-value
Baseline(default) - REWOSA(default)	0.12 (large)	6.772e-06
Baseline(default) - Baseline(SBST21)	0.3325 (medium)	0.01793
REWOSA(default) - REWOSA(SBST21)	0.54 (negligible)	0.6693
Baseline(SBST21) - REWOSA(SBST21)	0.3025 (medium)	0.02841

to the default setup. However, if we compare the different setups for REWOSA, we see that REWOSA performs similarly on each of the setups.

Table 5.2: Jaccard similarity

Generator - setup	# data points	Average	Interquartile Range
Baseline - SBST21	28	0.127	0.035
REWOSA - Default	71	0.102	0.06
REWOSA - SBST21	37	0.125	0.107

To get a better insight into how useful the different failing test cases per run are, we look into the Jaccard similarity between failing test cases within the same run. This means that if there is no or only one failing test case within a run, we can not calculate the Jaccard similarity, *e.g.,* baseline with the default setup only had one failing test per run maximum. In the appendix, tables A.5, A.6 and A.7 present a full overview of the similarity values, whereas table 5.2 gives an overview of the similarity.

We identify that the REWOSA has a lot of low with a couple of high similarity values, meaning there are a lot of non-similar failing roads and a couple of very similar failing roads. On the other hand the baseline has more centrally grouped similarity values, meaning they range from being slightly similar to similar failing roads. This indicates that, on average, REWOSA has a slightly lower similarity value compared to the baseline. However, we do see that the similarity values from the baseline are more grouped together, as the baseline's interquartile range is lower than the REWOSA. This means that the failing roads generated by REWOSA can generate slightly more diverse failing test cases, but there is a bigger range in diversity with REWOSA compared to the baseline.

RQ_2: How well does REWOSA perform compared to the baseline?

We have seen that the REWOSA can create more failing test cases that are also slightly more diverse than the baseline, but the diversity of the failing test cases generated by REWOSA

has a larger range compared to the baseline. With the help of statistical tests, we can conclude that REWOSA performs significantly better than the baseline.

An overview of the raw data used for the figures of this research question can be found in appendix A.

5.3 RQ3: Overhead of real-world seeding

When we look at figure 5.5, we can see two values on the x-axis, where the value 'Real' is the actual time it took to perform the simulations. The value 'Simulated' is the time that was spent inside the simulation. All these runs have a total time budget of 2 hours, corresponding to 7200 seconds.

If we take a look at our implementation REWOSA, depicted in figure 5.5, with the SBST21 and Default setups, we spent approximately 3082 and 3565 seconds on average on simulation time, respectively. That means that over half of the time budget is spent on creating and validating the test cases. As the time budget is 7200, a total of 4118 and 3635 seconds is spent on average on creating and validating the roads for the different setups. If we compare this to the baseline with the same setups, we spent approximately 6234 and 6327 seconds on average on simulation time, respectively. This means that, on average, 966 and 873 seconds are spent on creating and validating the roads for the different setups. This shows that the baseline uses about 86.6-87.9% of the time budget for simulating the test cases, whereas REWOSA uses only around 42.8-49.5% of the time budget for simulating the test cases. This shows that the baseline uses about double the amount of time for simulation compared to REWOSA.

In order to check if these results are significant, we performed statistical tests using a Wilcoxon rank sum test as well as the Vargha-Delaney's effect size measure for both the real and simulated time. A first glance on the results of the statistical tests for the real-time, in table 5.3, shows that the baseline performs significantly better than REWOSA. Interestingly, the baseline performs almost the same with the different setups, with a slight advantage for the SBST21 setup, whereas REWOSA performs a lot better with the SBST21 setup.

Now looking at the statistical test results for the simulated time, shown in table 5.4, we see again that the baseline significantly outperforms REWOSA. Here we also see that both the baseline and REWOSA perform significantly better using the SBST21 setup compared to the default setup.

$RQ_{3.1}$: How large is the preparation work required for REWOSA?

The preparation work required to get the real world roads starts of with selecting which roads to get. This can take anything from a minute, just picking a random road, up to ten or even more minutes, if doing research beforehand on what road to pick is preferred. Then comes the process of getting the road using MyMaps, which takes anywhere between three and five minutes. After that, the road needs to be downloaded and put somewhere in the project so it can be used by the generator. So for each road it takes at least five minutes time to get it to a point where it is usable for the generator, but it can also take a lot more time

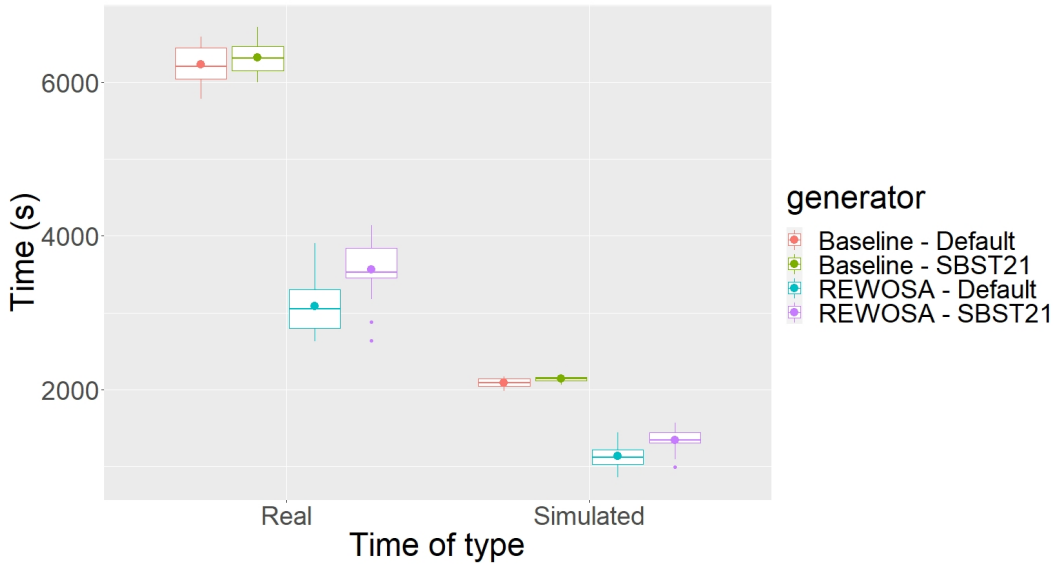

Figure 5.5: Overview of the different types of time for the simulations created by the different combinations of generator and setup. These figures display box-plots in which the outliers are shown with small dots and the average value is displayed by a big dot.

Table 5.3: This table shows results of statistical tests performed on the real time used by two different generator(setup) combinations using the Vargha-Delaney A measure for the effect size and the Wilcoxon rank sum for the p-value.

Generator(setup) comparison	Effect size measure	P-value
Baseline(default) - REWOSA(default)	1 (large)	1.451e-11
Baseline(default) - Baseline(SBST21)	0.405 (small)	0.3141
REWOSA(default) - REWOSA(SBST21)	0.17 (large)	0.0002
Baseline(SBST21) - REWOSA(SBST21)	1 (large)	1.451e-11

depending on how much research is put in the preparation of finding the roads.

$RQ_{3.2}$: How does the creation and validation time compare between the baseline and REWOSA?

As we have shown with the results, the baseline spends approximately 25% of the time REWOSA uses to create and validate each test cases. REWOSA uses over half of the time budget for the creation and validation of the roads, whereas the baseline uses just under 15% of the time budget for it. If we now come back to the main research question:

RQ_3: What is the overhead of real-world road seeding?

Table 5.4: This table shows results of statistical tests performed on the simulated time used by two different generator(setup) combinations using the Vargha-Delaney A measure for the effect size and the Wilcoxon rank sum for the p-value.

Generator(setup) comparison	Effect size measure	P-value
Baseline(default) - REWOSA(default)	1 (large)	1.451e-11
Baseline(default) - Baseline(SBST21)	0.2625 (large)	0.009484
REWOSA(default) - REWOSA(SBST21)	0.145 (large)	5.208e-05
Baseline(SBST21) - REWOSA(SBST21)	1 (large)	1.451e-11

With the use of the sub-questions we can conclude that real-world road seeding has a big overhead when it comes to preparation work to gather the roads. On top of that, the creation and validation of the real-world roads also takes a lot longer, increasing the overhead even more. This overhead increase is caused by the complexity of the roads, which then also transitions into the time to simulate roads. This additional simulation time for real-world roads concludes to total overhead when using real-world seeding.

This means that there is the trade-off between complexity of the roads and overhead, the more complex the roads are, the bigger the overhead.

A full overview of the time data is shown in appendix A.

5.4 Discussion

5.4.1 Road properties

When we look at the length of the real-world roads and the randomly generated roads, one of the big differences we see is the length of the roads. The randomly generated roads are a lot shorter than the real-world roads. These shorter roads could be caused by using the Deepjanus seed generator as a base for the randomly generated roads. The Deepjanus seed generator has multiple variables that can be tuned to improve the generator for different test setups. In our case, we use the preset values of this generator.

If we compare this to REWOSA and the real-world roads, we extract the real-world coordinates and fit them to the grid size given at the start of the test run. This means the roads will fill up a big part of the grid. Compared to the randomly generated roads, there is a chance that that road is only positioned in one corner of the grid due to how it is initialized, whereas the real-world roads are spread across the whole grid.

The road complexity comparison shows that the randomly generated roads are able to create roads with about half the number of left and right turns that the real-world roads have, but the size is 3 to 6 times as small. Because the Deepjanus seed generator does not utilize the full grid, resulting in the randomly generated road being 3 to 6 times smaller, there is a large room for improvement. If the Deepjanus seed generator was able to improve by a

large amount and thus be more optimized for the grid, these random generated roads could reach the same if not a higher complexity as the real-world roads.

Since the real-world roads are longer and therefore contain more left and right turns, they can create more failing test cases. This suggests that increasing the complexity will lead to a higher number of failing test cases. However, when we look at the ratio between road length and the number of turns and straights, the randomly generated roads seem to already have more turns and straights for the size of the roads they create. This means that if the number of failures is correlated to the number of turns that a road has, which also correlates to the length of the road, the baseline will be able to perform similarly if not better than REWOSA. But if the number of failures is related to the ratio between road length and turn segments, we would already expect a similar number of failures between the baseline and REWOSA, which is not the case. We can say that generating shorter roads will lead to a smaller point of failure, whereas longer roads will create a larger point of failure. However, it is difficult to generate longer valid roads for a random generator. Using real-world roads as the seed aids the search process to achieve longer valid roads.

One thing we do notice is that the diversity in valid roads for REWOSA with the SBST21 setup is larger than the diversity in valid roads for the other generator-setup combinations. When we look closely at the type of invalid roads we identified, most of them are due to self-intersecting roads. We also see in the results that the roads generated by REWOSA with the SBST21 setup are slightly shorter than those generated with the default setup. On top of that, we also see that REWOSA with the SBST21 setup has a bit more turns than REWOSA with the default setup. So, the roads are slightly shorter and contain a bit more turns. This, in return, could drive the generator to create more invalid tests, as more turns on shorter roads have a higher chance of intersecting with themselves.

5.4.2 Applicability and effectiveness

On the point of failures and performance, we have seen in the statistical tests that the baseline performed better with the SBST21 setup compared to the default setup. This means that the lower tolerance level with the max speed generated more failures than the higher tolerance with no speed limit. This causes us to believe that the roads created by the baseline do not give enough room for the car to get up to higher speeds for it to cause failures with the default setup. We can see this from the complexity of the randomly generated roads. Most of these roads are 3 to 6 times shorter than real-world roads, whereas the number of turns is only half. This means that there are many more turns in ratio to the size of the road, which gives the impression that there is not enough space to get the car up to speed before the next turn.

If we compare the two different setups with REWOSA, we identify that they perform very similar. This means that no speed limit with higher tolerance performs similarly to the lower tolerance with the speed limit. This could be the case because these roads have more space for the car to get up to speed. However, it could also be the case that the initial population contains roads leading to a failing scenario or close to generating a failing test case for both setups. That means the failures generated on the same setups are generated from the same roads in the initial population.

For the similarity, we have seen that the baseline creates slightly more similar failing test cases compared to REWOSA. This could stem from the mutation part of the algorithms, as the mutation in the baseline is done in a different way to the mutation in REWOSA. Since the real-world roads contain more points, and thereby we adjust/add/remove more points when we mutate the roads, the difference between the original and mutated roads might also be more significant. However, the percentage of the road that gets mutated by both algorithms is needed to get a better idea on if this difference in mutation plays a role in the diversity of failing test cases, which is something that could be looked into in future work.

5.4.3 Cost and Execution loads

In order to have an initial population for the real-world roads, we need to find roads to use within this initial population. This is the first point of overhead, as finding a usable road can take anywhere from 1 minute up to 10 minutes or even more. A majority of the selected roads may lead to the creation of invalid roads in the initial population. Having too many invalid roads within the initial population can cause unwanted results, as it cannot simulate many roads. Therefore it is also helpful to test the roads beforehand to see if they are valid or not, so one does not end up with a population of just invalid roads. So, finding valid real-world roads is a task that can increase the overhead quickly. On top of that, the road extraction process from MyMaps takes about 3 to 5 minutes. So if having 50 or more usable roads is desired, gathering them and testing them could already take more than a day.

From the simulation times, we see that the real-world roads consume considerably more time in the simulation compared to the randomly generated roads. The main factor here is that within the validation part of the algorithm, there is a nested for-loop over the segments (not the same as the segments talked about in the complexity) of the road. Since real-world roads are longer and more complex, they have more road segments. Dealing with a nested for-loop over all these road segments will cause the validation step containing this nested for-loop to increase quadratically in time. This can also be seen in the results, where the baseline uses approximately 25% of the time that REWOSA uses to create and validate their roads. This results in REWOSA using over half of the time budget to create and validate roads, whereas the baseline only uses 15% of their time budget to create and validate roads. This results in the baseline being able to create more roads in the set time budget then REWOSA. Due to this higher complexity, the loading time for the roads into the simulator will also take up more time. The simulation time itself is also higher due to the size of the real-world roads being 3 to 6 times as long. This causes the trade-off between complexity and overhead, higher complexity results in more overhead. However, we have seen that these complex roads can generate more failures with fewer roads in the same amount of time, which causes us to believe that this extra overhead, which stems from the higher complexity of roads, is worth it.

Chapter 6

Threats to validity

Threats to *construct validity* stem from the relation between theory and experimentation. The comparison between the different generator-setup combinations is based on well-established metrics, such as simulation time, failures, and validity. These metrics give a fair estimation of the efficiency (simulation time) and effectiveness (failures, validity, etc.) of the different generator-setup combinations.

Threats to *internal validity* concerns factors that influence our results. In order to combat the randomness resulting from using a genetic algorithm, we repeated each execution 20 times. We provide the median, mean and interquartile ranges for these runs. We also used the same tool for each of our runs, and each run of their specific generator-setup combination is performed in the same way.

Threats to *conclusion validity* regard the relationship between treatment and outcome. Next to providing the results of our experiments, we provide a statistical comparison between the different configurations. We used both the Wilcoxon rank sum test and the Vargha-Delaney A measure to highlight whether our results are statistically significant or not.

Threats to *external validity* consider the generalisation of our results. Although we hand-picked the real-world roads used for our experiment, we balanced out the type of roads selected, and the approach is applicable to any other roads.

Chapter 7

Conclusion and Future Work

Due to the downsides of cost and danger of testing self-driving cars on real roads, simulation-based testing has provided a rapidly emerging alternative. Simulation-based testing provides a safe place to test the self-driving software. On top of that, it eases the testing process by the ability to provide different environments at a rapid pace.

In this thesis, we present REWOSA, a new algorithm to test the lane-keeping assistant. There have been many algorithms before that test the lane-keeping assistant by using segments, points, or even frames to build up roads for the simulation. However, most of these roads are generated randomly and might not represent real-world roads that self-driving cars need to drive on. Some algorithms use real-world roads, like using police reports to recreate the actual roads[14], but no algorithm uses Google Maps to extract information, which is a lot more accessible and easy to extract. Therefore, we introduce REWOSA, an automated test generator for self-driving cars which uses real-world roads extracted from Google Maps to create scenarios for the simulations, and aims to use these real-world scenarios to depart the car from its lane. This algorithm is based on a state-of-the-art algorithm called FITEST, where we use multi-objective test generation and extent it by using mutation.

The contributions of this research are as follows: (i) providing a novel approach to test lane-keeping assistant for self-driving cars by using real-world roads, (ii) implementing this approach, and(iii) evaluating our approach against a state-of-the-art genetic algorithm as baseline using the BeamNG AI self-driving software.

When comparing our approach REWOSA with the baseline regarding fault detection, we see that REWOSA is able to provide an average of around 2 faults per run, whereas the baseline can only find 0 to just above 1 fault on average. Besides, we see that REWOSA creates 3 to 6 times longer roads containing an increased number of turns. These longer roads and more turns help to increase the chance of detecting a fault, leading to the higher average faults detection by REWOSA. In return, REWOSA brings a large overhead compared to the baseline due to the additional time-taking process of gathering roads from Google Maps and validating them. This validation process is larger due to a nested for-loop and the number of segments in the real-world roads being much larger than that of the randomly generated roads, which results in quadratically increasing the time needed to validate the real-world roads. Even though the overhead is larger for the real-world roads, we have seen that the overhead is worth it, as we detect more faults in the same time span.

In the future, we plan to extend the road gathering part by trying to create an automated process in order to find and extract the real-world roads from Google Maps, so they can directly be used in the road generator. This will help to speed up the real-world road finding and extracting process and thus decrease the overhead. On top of that, we want to conduct a more extensive comparison by running more tests.

Bibliography

[1] Ahmed AbdelHamed, Girma Tewolde, and Jaerock Kwon. Simulation framework for development and testing of autonomous vehicles. In *2020 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS)*, pages 1–6. IEEE, 2020.

[2] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing autonomous cars for feature interaction failures using many-objective search. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 143–154. IEEE, 2018.

[3] J Harold Ahlberg, Edwin Norman Nilson, and Joseph Leonard Walsh. *The Theory of Splines and Their Applications: Mathematics in Science and Engineering: A Series of Monographs and Textbooks, Vol. 38*, volume 38. Elsevier, 2016.

[4] Hesham Alghodhaifi and Sridhar Lakshmanan. Simulation-based model for surrogate safety measures analysis in automated vehicle-pedestrian conflict on an urban environment. In *Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure 2020*, volume 11415, page 1141504. International Society for Optics and Photonics, 2020.

[5] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[6] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 63–74, 2016.

[7] J Anthony Capon. Elementary statistics for the social sciences: Study guide, 1991.

[8] Edwin Catmull and Raphael Rom. A class of local interpolating splines. In *Computer aided geometric design*, pages 317–326. Elsevier, 1974.

[9] Rory Cellan-Jones. Uber's self-driving operator charged over fatal crash, Sep 2020. URL https://www.bbc.com/news/technology-54175359.

[10] Sikai Chen, Yue Leng, and Samuel Labi. A deep learning algorithm for simulating autonomous driving considering prior knowledge and temporal information. *Computer-Aided Civil and Infrastructure Engineering*, 35(4):305–321, 2020.

[11] Pouria Derakhshanfar, Xavier Devroey, Gilles Perrouin, Andy Zaidman, and Arie van Deursen. Search-based crash reproduction using behavioural model seeding. *Software Testing, Verification and Reliability*, 30(3):e1733, 2020.

[12] Marius Dupuis, Martin Strobl, and Hans Grezlikowski. Opendrive 2010 and beyond–status and future of the de facto standard for the description of road networks. In *Proc. of the Driving Simulation Conference Europe*, pages 231–242, 2010.

[13] Gordon Fraser and Andrea Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 121–130, 2012. doi: 10.1109/ICST.2012.92.

[14] Alessio Gambi, Tri Huynh, and Gordon Fraser. Generating effective test cases for self-driving cars from police reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 257–267, 2019.

[15] Alessio Gambi, Marc Mueller, and Gordon Fraser. Automatically testing self-driving cars with search-based procedural content generation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 318–328, 2019.

[16] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2009.

[17] Radoslav Ivanov, Taylor J Carpenter, James Weimer, Rajeev Alur, George J Pappas, and Insup Lee. Case study: verifying the safety of an autonomous racing car with a neural network controller. In *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, pages 1–7, 2020.

[18] Nidhi Kalra and Susan M. Paddock. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transportation Research Part A: Policy and Practice*, 94:182–193, 2016. ISSN 0965-8564. doi: https://doi.org/10.1016/j.tra.2016.09.010. URL https://www.sciencedirect.com/science/article/pii/S0965856416302129.

[19] Myounghoe Kim, Seongwon Lee, Jaehyun Lim, Jongeun Choi, and Seong Gu Kang. Unexpected collision avoidance driving strategy using deep reinforcement learning. *IEEE Access*, 8:17243–17252, 2020. doi: 10.1109/ACCESS.2020.2967509.

[20] Florian Klück, Martin Zimmermann, Franz Wotawa, and Mihai Nica. Genetic algorithm-based test parameter optimization for adas system testing. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pages 418–425. IEEE, 2019.

[21] Laurent Magnier and Fariborz Haghighat. Multiobjective optimization of building design using trnsys simulations, genetic algorithm, and artificial neural network. *Building and Environment*, 45(3):739–746, 2010.

[22] Sivabalan Manivasagam, Shenlong Wang, Kelvin Wong, Wenyuan Zeng, Mikita Sazanovich, Shuhan Tan, Bin Yang, Wei-Chiu Ma, and Raquel Urtasun. Lidarsim: Realistic lidar simulation by leveraging the real world. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11167–11176, 2020.

[23] John McCarty. Computer controlled cars. *Obtenido de Universidad de Stan-ford: http://www-formal. stanford. edu/jmc/progress/cars/cars. html*, 1968.

[24] Sky McKinley and Megan Levine. Cubic spline interpolation. *College of the Redwoods*, 45(1):1049–1060, 1998.

[25] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[26] Melanie Mitchell. Genetic algorithms: An overview. In *Complex.*, volume 1, pages 31–39. Citeseer, 1995.

[27] Mahshid Helali Moghadam, Markus Borg, and Seyed Jalaleddin Mousavirad. Deeper at the sbst 2021 tool competition: Adas testing using multi-objective search. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 40–41. IEEE, 2021.

[28] Dan Negrut, Radu Serban, Asher Elmquist, Dylan Hatch, Eric Nutt, and Phil Sheets. Autonomous vehicles in the cyberspace: Accelerating testing via computer simulation. Technical report, SAE Technical Paper, 2018.

[29] Vuong Nguyen, Stefan Huber, and Alessio Gambi. Salvo: Automated generation of diversified tests for self-driving cars from existing maps. In *2021 IEEE International Conference on Artificial Intelligence Testing (AITest)*, pages 128–135, 2021. doi: 10.1109/AITEST52744.2021.00033.

[30] Mitchell Olsthoorn, Pouria Derakhshanfar, and Xavier Devroey. An application of model seeding to search-based unit test generation for gson. In *International Symposium on Search Based Software Engineering*, pages 239–245. Springer, 2020.

[31] Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. Sbst tool competition 2021. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 20–27. IEEE, 2021.

[32] Bryan Pietsch. 2 killed in driverless tesla car crash, officials say, Apr 2021. URL https://www.nytimes.com/2021/04/18/business/tesla-fatal-crash-texas.html.

[33] Dean A Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural computation*, 3(1):88–97, 1991.

[34] Ryan Randazzo. Waymo to start driverless ride sharing in phoenix area this year, Jan 2018. URL https://eu.azcentral.com/story/money/business/tech/2018/01/30/waymo-start-driverless-ride-sharing-phoenix-area-year/1078466001/.

[35] Mustafa Saraoglu, Andrey Morozov, and Klaus Janschek. Mobatsim: Model-based autonomous traffic simulation framework for fault-error-failure chain analysis. *IFAC-PapersOnLine*, 52(8):239–244, 2019.

[36] Sanjana Shivdas and Tim Kelly. Toyota halts all self-driving e-palette vehicles after olympic village accident, Aug 2021. URL https://www.reuters.com/business/autos-transportation/toyota-halts-all-self-driving-e-pallete-vehicles-after-olympic-village-accident-2021-08-27/.

[37] SN Sivanandam and SN Deepa. Genetic algorithms. In *Introduction to genetic algorithms*, pages 15–37. Springer, 2008.

[38] Tong Duy Son, Ajinkya Bhave, and Herman Van der Auweraer. Simulation-based testing framework for autonomous driving development. In *2019 IEEE International Conference on Mechatronics (ICM)*, volume 1, pages 576–583. IEEE, 2019.

[39] The Tesla Team. All tesla cars being produced now have full self-driving hardware, Oct 2016. URL https://www.tesla.com/blog/all-tesla-cars-being-produced-now-have-full-self-driving-hardware.

[40] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[41] Alexander Von Bernuth, Georg Volk, and Oliver Bringmann. Rendering physically correct raindrops on windshields for robustness verification of camera-based object recognition. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 922–927. IEEE, 2018.

[42] Ralph Weissnegger, Christian Kreiner, Markus Pistauer, Kay Römer, and Christian Steger. Sharc-simulation and verification of hierarchical embedded microelectronic systems. *Procedia Computer Science*, 109:392–399, 2017.

Appendix A

Data

Table A.1 displays the results from the REWOSA generator combined with the Default setup. Table A.2 gives an overview of the REWOSA generator using the SBST21 setup. Then table A.3 and A.4 give an overview of the baseline generator with the Default and SBST21 setup respectively.

Each of these tables gives an shows the results for each run separately as well a the mean and median of all the runs. The information in the tables is as follows; (i) Name, differentiating between the different runs; (ii) Generated, the number of test cases generated during the run; (iii) Valid, the number of valid test cases; (iv) Invalid, the number of invalid test cases; (v) Passed, the number of test cases that passed; (vi) Failed, the total number of test cases that failed; (vii) Failed start, the number of test cases that failed on first creation, without mutation; (viii) Error, the number of test cases that returned an error during the simulation. Then the bottom two lines in the tables display the Mean, being the average of all the above listed test runs, and Median, being the center value of the values listed above, when sorted from smallest to largest.

A.1 Similarity

The similarity values are a list of tuples, where the first value represent a tuple of the test cases that are being compared. This number is just an index of a failing test during that run, meaning (1,3) would represent a tuple between the failing tests with index 1 and 3. The second value is the actual similarity value, which represents how much of the 2 sets is similar. The lower similarity values highlight that there is little to no similarity between the roads, whereas a high similarity value means that there is a lot of similarity.

A.2 Simulation time

Table A.1: REWOSA with default setup

Name	Generated	Valid	Invalid	Passed	Failed	Failed start	Error
Run 1	26	16	10	9	2	0	4
Run 2	26	20	6	11	5	1	3
Run 3	29	22	7	11	6	0	5
Run 4	22	17	5	14	0	0	2
Run 5	28	21	7	10	5	2	5
Run 6	32	23	9	10	3	0	9
Run 7	23	18	5	15	3	2	0
Run 8	26	18	8	9	3	1	6
Run 9	33	27	6	13	4	1	10
Run 10	13	10	3	6	0	0	4
Run 11	22	18	4	13	1	0	4
Run 12	24	18	6	10	1	0	6
Run 13	12	10	2	8	0	0	1
Run 14	17	11	6	7	3	1	0
Run 15	26	20	6	10	2	1	7
Run 16	14	11	3	6	1	0	3
Run 17	27	22	5	10	6	0	6
Run 18	26	20	6	9	2	1	9
Run 19	16	12	4	7	1	1	3
Run 20	12	10	2	7	0	0	2
Mean	22.7	17.2	5.5	9.75	2.4	0.55	4.45
Median	25	18	6	10	2	0	4

Table A.2: REWOSA with SBST21 setup

Name	Generated	Valid	Invalid	Passed	Failed	Failed start	Error
Run 1	28	20	8	13	2	1	4
Run 2	26	19	7	9	5	0	4
Run 3	15	15	0	11	1	0	3
Run 4	20	14	6	9	1	0	3
Run 5	28	19	9	12	2	0	4
Run 6	24	14	10	11	1	1	1
Run 7	15	13	2	7	3	1	2
Run 8	15	10	5	4	2	0	3
Run 9	22	13	9	8	0	0	5
Run 10	23	18	5	11	2	0	4
Run 11	22	15	7	8	1	0	5
Run 12	20	16	4	8	3	0	4
Run 13	18	15	3	12	0	0	3
Run 14	34	24	10	14	4	1	5
Run 15	27	19	8	12	4	0	3
Run 16	21	14	7	11	1	1	1
Run 17	23	17	6	11	2	1	1
Run 18	22	14	8	9	3	1	1
Run 19	27	21	6	12	2	0	6
Run 20	11	11	0	5	1	0	4
Mean	22.05	16.05	6	9.85	2	0.35	3.3
Median	22	15	6.5	11	2	0	3.5

Table A.3: Baseline with default setup

Name	Generated	Valid	Invalid	Passed	Failed	Failed start	Error
Run 1	154	120	34	103	0	0	17
Run 2	124	91	33	77	0	0	14
Run 3	200	136	64	126	0	0	10
Run 4	162	109	53	92	1	0	16
Run 5	195	130	65	115	0	0	15
Run 6	131	95	36	81	0	0	14
Run 7	119	92	27	83	0	0	9
Run 8	149	103	46	88	1	0	14
Run 9	131	97	34	84	0	0	24
Run 10	152	110	42	85	0	0	24
Run 11	161	115	46	105	0	0	10
Run 12	150	107	43	96	0	0	11
Run 13	128	92	36	70	0	0	22
Run 14	184	123	61	105	0	0	17
Run 15	187	140	47	124	0	0	16
Run 16	122	91	31	70	0	0	21
Run 17	133	104	29	83	0	0	21
Run 18	152	117	35	95	0	0	22
Run 19	198	142	56	130	0	0	12
Run 20	205	146	59	126	0	0	20
Mean	156.85	113	43.85	96.9	0.1	0	16.45
Median	152	109.5	42.5	93.5	0	0	16

Table A.4: Baseline with SBST21 setup

Name	Generated	Valid	Invalid	Passed	Failed	Failed start	Error
Run 1	171	134	37	125	1	0	8
Run 2	152	112	40	94	2	0	16
Run 3	143	103	40	91	0	0	12
Run 4	175	128	47	101	0	0	27
Run 5	127	88	39	78	0	0	10
Run 6	126	95	31	75	4	0	16
Run 7	136	101	35	87	3	1	11
Run 8	119	91	28	77	0	0	14
Run 9	138	95	43	87	0	0	8
Run 10	157	111	46	103	2	0	6
Run 11	152	116	36	102	4	1	10
Run 12	201	141	60	91	0	0	50
Run 13	179	129	50	114	2	0	12
Run 14	133	94	39	78	5	0	11
Run 15	184	138	46	110	0	0	28
Run 16	125	94	31	77	0	0	17
Run 17	125	90	35	75	0	0	14
Run 18	123	92	31	73	0	0	18
Run 19	162	115	47	104	0	0	11
Run 20	132	101	31	86	0	0	15
Mean	148	108.4	39.6	91.4	1.15	0.1	15.7
Median	140.5	102	39	89	0	0	13

Table A.5: Jaccard similarity Baseline with SBST21 setup

Run	Similarity values
Run 2	((0, 1), 0.169)
Run 6	((0, 1), 0.140), ((0, 2), 0.127), ((0, 3), 0.120), ((1, 2), 0.280), ((1, 3), 0.146), ((2, 3), 0.133)
Run 7	((0, 1), 0.065), ((0, 2), 0.096), ((1, 2), 0.147)
Run 10	((0, 1), 0.160)
Run 11	((0, 1), 0.059), ((0, 2), 0.065), ((0, 3), 0.075), ((1, 2), 0.111), ((1, 3), 0.128), ((2, 3), 0.113)
Run 13	((0, 1), 0.047)
Run 14	((0, 1), 0.121), ((0, 2), 0.150), ((0, 3), 0.133), ((0, 4), 0.127), ((1, 2), 0.15), ((1, 3), 0.134), ((1, 4), 0.125), ((2, 3), 0.137), ((2, 4), 0.122), ((3, 4), 0.163)

Table A.6: Jaccard similarity REWOSA with Default setup

Run	Similarity values
Run 1	((0, 1), 0.067)
Run 2	((0, 1), 0.062), ((0, 2), 0.073), ((0, 3), 0.070), ((0, 4), 0.071), ((1, 2), 0.107), ((1, 3), 0.115), ((1, 4), 0.142), ((2, 3), 0.114), ((2, 4), 0.130), ((3, 4), 0.139)
Run 3	((0, 1), 0.104), ((0, 2), 0.057), ((0, 3), 0.059), ((0, 4), 0.075), ((0, 5), 0.111), ((1, 2), 0.048), ((1, 3), 0.065), ((1, 4), 0.074), ((1, 5), 0.098), ((2, 3), 0.071), ((2, 4), 0.079), ((2, 5), 0.063), ((3, 4), 0.080), ((3, 5), 0.067), ((4, 5), 0.081)
Run 5	((0, 1), 0.068), ((0, 2), 0.067), ((0, 3), 0.067), ((0, 4), 0.071), ((1, 2), 0.117), ((1, 3), 0.115), ((1, 4), 0.123), ((2, 3), 0.114), ((2, 4), 0.130), ((3, 4), 0.127)
Run 6	((0, 1), 0.127), ((0, 2), 0.127), ((1, 2), 0.243)
Run 7	((0, 1), 0.070), ((0, 2), 0.078), ((1, 2), 0.225)
Run 8	((0, 1), 0.129), ((0, 2), 0.060), ((1, 2), 0.086)
Run 9	((0, 1), 0.056), ((0, 2), 0.052), ((0, 3), 0.074), ((1, 2), 0.123), ((1, 3), 0.194), ((2, 3), 0.137)
Run 14	((0, 1), 0.091), ((0, 2), 0.076), ((1, 2), 0.131)
Run 15	((0, 1), 0.242))
Run 17	((0, 1), 0.128), ((0, 2), 0.109), ((0, 3), 0.064), ((0, 4), 0.158), ((0, 5), 0.137), ((1, 2), 0.103), ((1, 3), 0.066), ((1, 4), 0.155), ((1, 5), 0.123), ((2, 3), 0.067), ((2, 4), 0.117), ((2, 5), 0.128), ((3, 4), 0.071), ((3, 5), 0.066), ((4, 5), 0.137)
Run 18	((0, 1), 0.063)

Table A.7: Jaccard similarity REWOSA with SBST21 setup

Run	Similarity values
Run 1	((0, 1), 0.055)
Run 2	((0, 1), 0.115), ((0, 2), 0.083), ((0, 3), 0.129), ((0, 4), 0.086), ((1, 2), 0.077), ((1, 3), 0.121), ((1, 4), 0.087), ((2, 3), 0.099), ((2, 4), 0.109), ((3, 4), 0.089)
Run 5	((0, 1), 0.068)
Run 7	((0, 1), 0.145), ((0, 2), 0.156), ((1, 2), 0.111)
Run 8	((0, 1), 0.296)
Run 10	((0, 1), 0.071)
Run 12	((0, 1), 0.125), ((0, 2), 0.104), ((1, 2), 0.095)
Run 14	((0, 1), 0.096), ((0, 2), 0.172), ((0, 3), 0.072), ((1, 2), 0.087), ((1, 3), 0.033), ((2, 3), 0.054)
Run 15	((0, 1), 0.222), ((0, 2), 0.217), ((0, 3), 0.230), ((1, 2), 0.232), ((1, 3), 0.223), ((2, 3), 0.190)
Run 17	((0, 1), 0.063)
Run 18	((0, 1), 0.201), ((0, 2), 0.033), ((1, 2), 0.035)
Run 19	((0, 1), 0.240)

Table A.8: Simulation time REWOSA with default setup

Name	Real time execution (sec)	Simulated time execution (sec)
Run 1	2825.233	1010.750288
Run 2	3357.422	1223.124051
Run 3	3047.642	1116.740793
Run 4	2992.437	1116.513794
Run 5	3053.391	1118.829292
Run 6	3368.016	1190.4123
Run 7	3814.392	1409.841309
Run 8	2733.641	998.2272891
Run 9	3900.986	1440.42806
Run 10	2817.765	1069.003296
Run 11	3286.33	1222.33505
Run 12	2920.266	1069.606546
Run 13	2626.297	991.225292
Run 14	2650.954	852.5642798
Run 15	3202.187	1183.355797
Run 16	2641.53	994.8035409
Run 17	3388.954	1256.406799
Run 18	3261.125	1209.289298
Run 19	3053.904	1156.210547
Run 20	2695.532	1026.228041
Mean	3081.9002	1132.79478314
Median	3050.5165	1117.7850425

Table A.9: Simulation time REWOSA with SBST21 setup

Name	Real time execution (sec)	Simulated time execution (sec)
Run 1	4142.279	1562.670058
Run 2	3460.987	1296.645802
Run 3	3449.968	1304.454052
Run 4	3562.718	1353.598807
Run 5	3833.794	1443.422558
Run 6	3475.311	1318.787053
Run 7	3475.753	1316.300054
Run 8	2640.656	992.1547896
Run 9	3483.549	1322.005307
Run 10	3894.941	1472.666306
Run 11	3288.329	1241.680553
Run 12	3175.204	1194.551548
Run 13	3705.375	1403.916556
Run 14	3973.062	1479.041557
Run 15	4092.205	1531.90931
Run 16	3488.345	1323.399053
Run 17	3771.077	1422.746054
Run 18	3643.733	1382.057059
Run 19	3873.408	1442.708059
Run 20	2876.454	1092.640293
Mean	3565.3574	1344.86774143
Median	3525.5315	1338.49893

Table A.10: Simulation time Baseline with default setup

Name	Real time execution (sec)	Simulated time execution (sec)
Run 1	6488.908	2076.210571
Run 2	6035.015	2080.250569
Run 3	6569.165	2166.165573
Run 4	6334.245	2146.931321
Run 5	6477.611	2144.839571
Run 6	6046.521	2023.103318
Run 7	5868.662	2011.896317
Run 8	5900.169	2006.736316
Run 9	6209.089	2139.484068
Run 10	6123.026	2071.145319
Run 11	6187.285	2038.605068
Run 12	6194.445	2094.517066
Run 13	5975.53	2046.959069
Run 14	6400.968	2140.004818
Run 15	6588.506	2160.614322
Run 16	5782.893	1984.239815
Run 17	6117.136	2079.039568
Run 18	6383.161	2151.812821
Run 19	6561.838	2145.425568
Run 20	6436.141	2086.664568
Mean	6234.0157	2089.58228075
Median	6201.767	2083.4575685

Table A.11: Simulation time Baseline with SBST21 setup

Name	Real time execution (sec)	Simulated time execution (sec)
Run 1	6714.808	2162.080325
Run 2	6345.264	2134.53432
Run 3	6324.466	2153.113319
Run 4	6539.292	2175.568321
Run 5	6285.181	2191.151323
Run 6	6110.578	2104.623568
Run 7	6137.163	2102.655567
Run 8	6304.826	2188.063322
Run 9	6235.811	2144.00882
Run 10	6416.037	2148.581066
Run 11	6429.108	2134.974573
Run 12	6552.199	2111.549572
Run 13	6529.907	2160.764319
Run 14	6131.62	2109.120066
Run 15	6523.584	2142.524321
Run 16	6152.025	2115.021317
Run 17	6097.595	2111.17232
Run 18	5998.548	2063.970066
Run 19	6449.469	2173.739072
Run 20	6259.477	2125.171069
Mean	6326.8479	2137.6193323
Median	6314.646	2138.749447

Appendix B

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

`REWOSA`: Real World Search Algorithm

OOB: Out Of Bounds

OBE: Out of Bounds Episode

SBST(21): Search-Based Software Testing, where 21 refers to the year 2021

AI: Artificial Intelligence

Lidar: Light detection and ranging