Lasse Slaar

Evolutionary Optimization of Simulation Environments for Benchmarking Robot Path Planning Controllers in ROS 2

FAKULTÄT FÜR
INFORMATIK

Intelligent Cooperative Systems
Computational Intelligence

Evolutionary Optimization of Simulation Environments for Benchmarking Robot Path Planning Controllers in ROS 2

Bachelor Thesis

Lasse Slaar

March 21, 2026

Supervisor: Sanaz Mostaghim

Advisor: Carlo Nübel

Lasse Slaar: *Evolutionary Optimization of Simulation Environments for Benchmarking Robot Path Planning Controllers in ROS 2*
Otto-von-Guericke Universität
Intelligent Cooperative Systems
Computational Intelligence
Magdeburg, 2026.

Abstract

In the era of increasing automation, the field of robotics continues to expand rapidly, making robust and reliable navigation algorithms increasingly important. Numerous scientific studies have compared different local trajectory planners. However, in most cases, the environments used for evaluation are manually designed. The objective of this thesis is to investigate whether an evolutionary approach can be used to automatically generate a set of benchmark maps that enables a progressively more differentiated comparison of multiple planners. The goal is to create maps such that, on an initial map, all planners perform similarly, while on subsequent maps their performance differences become increasingly pronounced. Ultimately, this approach aims to generate benchmark scenarios automatically, enabling systematic comparison of different planners as well as parameter tuning within a given planner. To evaluate this concept, a real-time simulation environment was implemented using ROS 2 and Gazebo. A Genetic Algorithm with two fitness functions was developed to explore the feasibility of this approach. Additionally, methods for reducing simulation runtime were investigated in order to make the evaluation of real-time simulations across many generations computationally feasible. The results demonstrate that both fitness functions can generate map sets suitable for controller comparison. Benchmark evaluations reveal that the first fitness produced slightly better results. However, both approaches successfully create the desired pattern of increasing performance differences across the generated maps.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

In recent years, digitalization and automation have become increasingly important in many industrial environments. In particular, internal logistics systems require efficient and reliable automation solutions to ensure productivity and maintain flexibility [3].

A key development in this field is the evolution of automated guided vehicles (AGVs) into autonomous mobile robots (AMRs). These robots are able to navigate through complex environments using modern sensing technologies and artificial intelligence [4].

As a result, autonomous mobile robots are employed in industrial production, urban logistics and search-and-rescue scenarios, where they perform tasks in environments that may be complex or initially unknown [5].

To operate in such environments, AMRs rely on robust navigation systems. These systems enable them to locate themselves, sense their surroundings and plan paths. Since efficiency is critical, planners must operate reliably and their algorithms must be highly optimized. An important question is which planner is suitable for different types of environments. To answer this, it is necessary to understand the strengths and weaknesses of these algorithms. Consequently, systematic comparisons of these algorithms are conducted.

However comparing navigation algorithms is a non-trivial task. It is necessary to select the environments in which the planners are evaluated, which are often manually designed [6, 7, 8], adjust the planners' navigation parameters, and define the metrics for comparison. Common metrics reported in the literature include path length, computation time, success rate or collision avoidance [6, 7, 8]. Since all of these factors must be considered, the comparison of multiple planners is a time-consuming task. Decisions regarding these factors may introduce bias, and consequently, conducting multiple comparisons across diverse and increasingly complex environments is challenging.

This thesis investigates whether evolutionary optimization can be employed to automatically generate sets of benchmark maps that reveal performance differences, strengths,

and weaknesses among multiple navigation planners. To achieve this, two distinct fitness functions are evaluated using a genetic algorithm. Furthermore, the challenges associated with assessing real-time simulation environments are examined.

1.2 Goals

The primary objective of this thesis is to investigate how genetic algorithms can be used to generate sets of maps that enable a meaningful comparison of previously selected local planners based on predefined performance metrics.

The secondary objective is to examine how to address the problem that evaluating a single individual in an evolutionary algorithm requires a significant amount of time due to the computational cost of simulation. In particular, the work explores how statistically significant results can be obtained within a reasonable time frame.

The following research questions are addressed in this thesis:

RQ1: How can an evolutionary algorithm be applied to automatically generate representative test scenarios for evaluating local planners in ROS 2?

RQ2: How do different fitness functions affect the ability of the EA defined in RQ1 to generate test scenarios that highlight the performance differences between local planners in ROS 2?

RQ3: How does the population size influence the effectiveness of the EA defined in RQ1 to generate test scenarios that highlight performance differences between local planners in ROS 2?

For this question, we define two hypotheses:

- **H0**: The population size has no statistically significant impact on the effectiveness and stability of the evolutionary algorithm in generating test scenarios for benchmarking local planners in ROS 2.
- **H1**: At least one population size has a statistically significant impact on the effectiveness and stability of the evolutionary algorithm in generating test scenarios for benchmarking local planners in ROS 2.

RQ4: How can the computational cost of repeated simulations be reduced when evolutionary algorithms require frequent evaluations?

1.3 Thesis structure

This thesis is structured as follows. First, the theoretical background on ROS 2 and evolutionary algorithms is introduced in chapter 2. Next, related work is reviewed in chapter 3, and the research gap addressed by this thesis is identified. The methodology is then described in chapter 4, including the two fitness functions used in the proposed approach. After that, the experimental setup and parameter choices are introduced in chapter 5. The results of the conducted experiments are then presented and analyzed. Finally, potential extensions are discussed in 6, particularly with regard to its limitations and possible improvements to the evolutionary algorithm.

2 Background

This chapter lays the technical foundation of the thesis. It introduces the Robot Operating System and explains the principles of robotic motion planning, including the distinction between global and local planners and the structure of the Navigation Stack.

In addition, the chapter presents evolutionary algorithms as the second core component of this work. Their fundamental principles, operators and algorithmic structure are discussed, followed by a description of the Python framework Pymoo [9], which is used to implement these algorithms.

2.1 ROS 2

The following section presents all the technical details of ROS 2 that are relevant for the practical implementation of the proposed approach. In particular, it focuses on the components required to simulate robotic navigation within a ROS 2 environment.

The Robot Operating System (ROS) is an open-source framework for developing robotic applications. Although its name suggests otherwise, ROS is technically a middleware that provides communication mechanisms, hardware abstraction, libraries and development tools.

Since 2022, ROS 2 [10] has been available as the successor to ROS. This development was necessary because the original ROS was not designed for industrial applications. Furthermore, ROS faced additional limitations, such as a single point of failure and insufficient support for multi-robot systems. To address the challenges of today's robotic systems, ROS 2 was designed. Given that ROS 2 is receiving increasing attention in modern research, this thesis adopts the newer version of ROS, ROS 2.

In ROS 2 the system is composed of distributed components called **nodes**, which communicate via well-defined interfaces known as topics. The communication architecture in ROS 2 follows a **publish/subscribe** model. Nodes can **publish** data to topics, making

it available to other nodes, or **subscribe** to receive the data published by other nodes [11].

Another fundamental concept of ROS 2 are **services**. While the communication over topics is asynchronous, a service provides synchronous communication and is implemented by a server. This server receives a request containing input data, executes the corresponding procedure and returns the result to the requesting node [11]. Services are typically used for **on-demand requests**.

A related concept is that of actions. Like services, actions allow nodes to execute requested procedures. However, unlike services, which block the resource until completion, actions provide the ability to receive progress feedback and can be interrupted or canceled by requests [11].

ROS 2 uses the **Data Distribution Service (DDS)** protocol for communication, a standard for distributed, real-time systems [2]. DDS enables ROS 2 to establish reliable and secure communication between nodes without requiring manual configuration of addresses or connections.

To determine which nodes are currently active, ROS 2 performs a process known as **discovery**. Each node broadcasts information about its presence to all other nodes on the network that share the same ROS domain, and notifies them when it is shutting down. The mapping of nodes to domains is defined by the ROS_DOMAIN_ID [11].

The **Driving Swarm framework** is a software infrastructure that supports the development and execution of experiments with multiple mobile robots. It provides a unified environment in which multi-robot experiments can be implemented, executed, monitored, and subsequently analyzed. The goal of the framework is to simplify the execution of reproducible experiments by providing reusable software components as well as standardized procedures for experiment management [12].

Driving Swarm supports both simulation environments and real robotic platforms, such as TurtleBot3 robots. This allows researchers to develop and test multi-robot behaviors in simulation before transferring them to real hardware, while using the same software architecture in both cases.

The simulated environment of Driving Swarm Infrastructure uses instances of Gazebo [13], a three-dimensional robotics simulator that provides realistic physics and sensor modeling for testing robot algorithms in complex virtual scenarios. By integrating Gazebo, Driving Swarm allows experiments to be executed in a controlled simulation before being deployed on real robots, ensuring consistency between virtual and physical traits.

2.1.1 Nav2

A useful framework to coordinate navigation in ROS 2 is Nav2 [2]. It consists of a collection of ROS 2 nodes that communicate via actions. The framework follows a modular architecture and supports extensibility through a plugin-based system.

The main architecture consists of these main components:

- BT Navigator Server
- Planner Server
- Controller Server
- Behavior Server

A visual image of the architecture can be seen in Figure 2.1.

All these components are so-called lifecycle nodes that are managed by a lifecycle manager. Lifecycle Nodes are Nodes that implement a lifecycle state-machine. This state-machine consists of 4 main states.

- Unconfigured
- Inactive
- Active
- Finalized

This is done so that a reliable transition is given when starting the program. With this, the current state of the system is more transparent and dependencies between nodes can be managed more easily [14, 15].

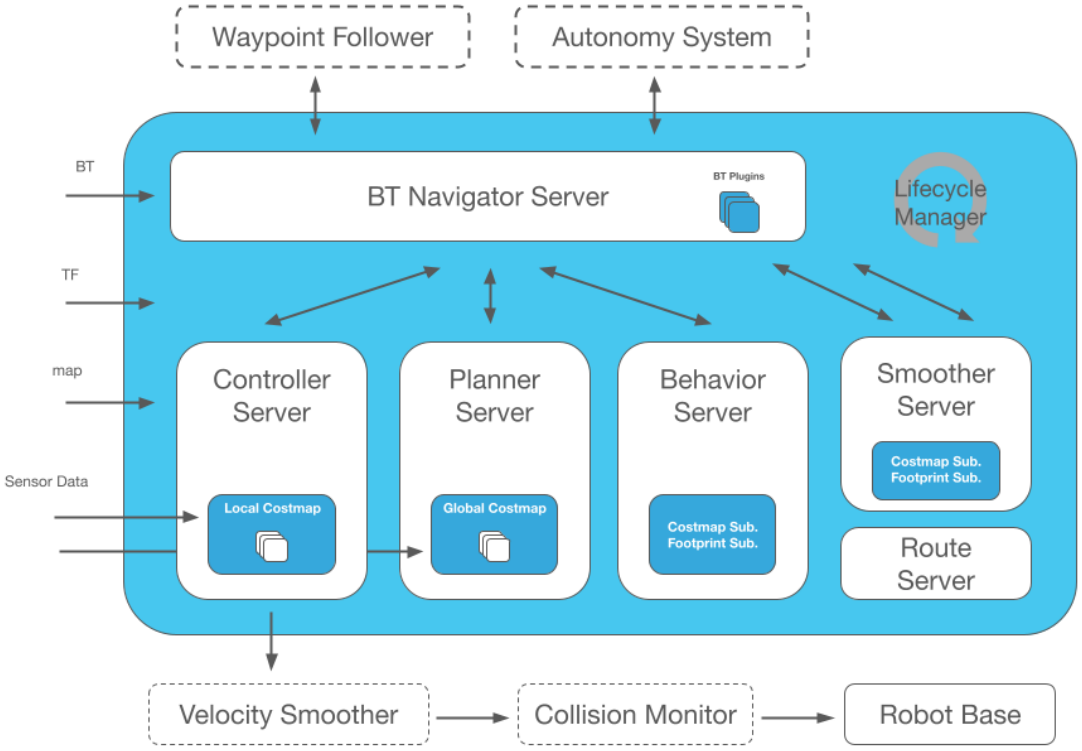

Figure 2.1: Nav2 - Architecture [1]

The Planner Server is responsible for global path planning. It processes planning requests and manages multiple planner plugins that compute feasible paths towards a given goal pose. The server operates on the global costmap and provides the resulting path to the Controller Server for execution [1]. In the example shown in Figure 2.2, the global planner is implemented using the A* algorithm [16].

The global costmap is provided to the Planner Server to compute a complete Path through the environment. It is a two-dimensional grid-based data structure that represents the entire environment. The costmap primarily contains information from the static map as the basis for the planner server to compute the initial path from the start position to the goal. This path is subsequently sent to the controller server for execution [1].

The Controller Server is responsible for computing velocity commands based on the global path and information from the local and global costmaps. Using this information, the controller generates motion commands that guide the robot along the planned path while avoiding obstacles. Nav2 provides several controller implementations that can be selected as plugins [1].

The local costmap is also a two-dimensional grid-based structure which is containing information about the immediate surroundings of the robot. It is typically generated from sensor data and is used by the controller server to calculate the next movement, ensuring safe and reactive navigation [1].

The Behavior Server provides additional behaviors that are executed when the controller is unable to compute a valid trajectory or when the robot is blocked by obstacles in the costmap. These behaviors are typically used as recovery actions during navigation [1]. In the behavior tree shown in Figure 2.2, the Behavior Server corresponds to the fallback branch on the right side of the tree.

The BT Navigator represents the central component of the navigation architecture and is ruling the navigation process using behavior trees. It provides actions such as *NavigateToPose* or *NavigateThroughPose*, which enable the robot to navigate from a start position to a specified goal [1]. The BT Navigator activates the navigation process and periodically provides feedback on the progress of the planner, controller or recovery services [2]. Based on the current state of the navigation task, it sequentially invokes the corresponding servers responsible for planning, control, or recovery behaviors.

Figure 2.2 shows an example Behavior Tree for a Nav2-based navigation application. Both the figure and the the accompanying description are closely based on [2].

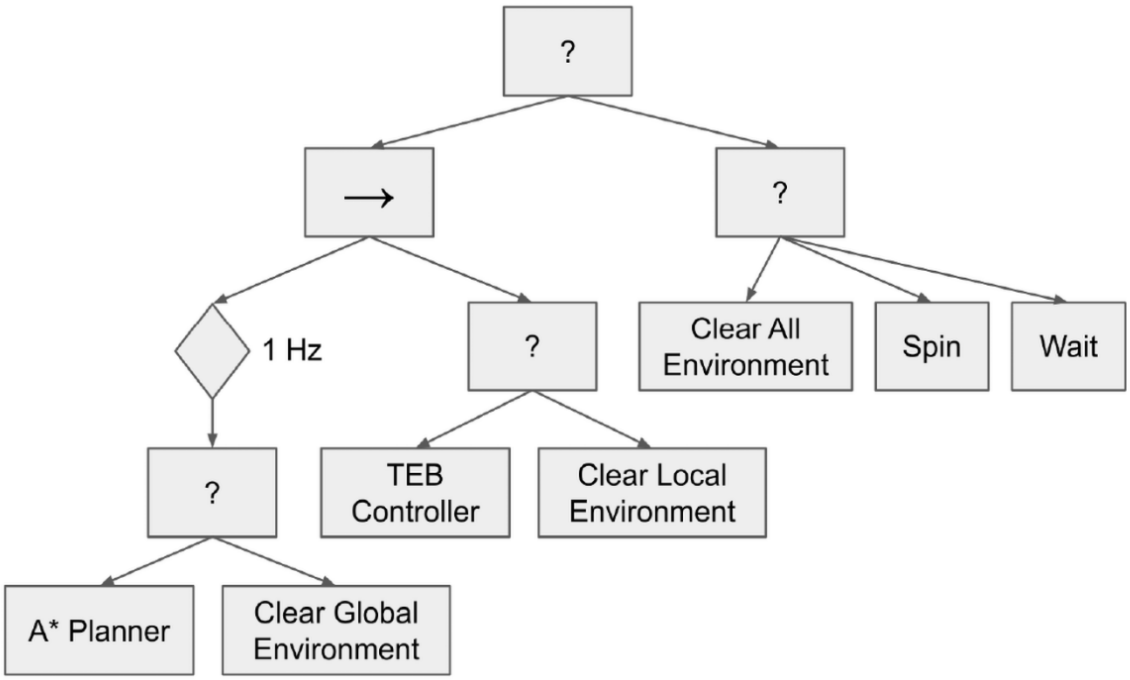

Figure 2.2: A simple behavior Tree for Nav2 Application from [2]

In this example, different types of nodes govern the behavior: The square with a '?' represents a fallback node, which is executed if its left child fails. The arrow represents a sequence node, executing its children from left to right. The diamond ticks its child at a frequency of 1 Hz.

The tree first ticks the Planner node (A* Planner) until a global path is successfully generated. If this fails, the fallback action clears the global environment. Subsequently, the TEB Controller node is executed to compute a local path; failure triggers the fallback action to clear the local environment. Finally, if the entire sequence fails, the environment is cleared and the system waits for the next tick [2].

This Behavior Tree is fully managed by the BT Navigator and integrates all previously discussed Nav2 components.

2.1.2 Planner

After introducing the architecture of Nav2, it is important to distinguish between global and local path planning. As described previously, the planner server computes a path

based on the global costmap, whereas the controller server generate a trajectory based on the global path and the local costmap.

In the following sections, we refer to the planning instance of the planner server as *global planner*, while the planning instance of the controller server is referred to as the *local planner*.

Global Planner

The process of navigating a robot to a specific goal can be divided into two main components. The first component, which is discussed in this section, is referred to as *global path planning*.

The objective of the global planner is to compute a cost-optimal path to a specified goal based solely on information about the global environment. In this stage, constraints related to the robot's dynamics, such as the maximal velocity or acceleration, are not considered [17].

To compute a cost-optimal path, it is necessary to define an appropriate cost function. The cost values are typically provided to the global planner by the global costmap. By defining the cost function accordingly, different objectives can be achieved. For example, the planner may compute the shortest path to the goal or prioritize a high path coverage [1].

Typically, global path planning algorithms are variations of Dijkstra's algorithm [18] or A^* search algorithm [16]. Nav2 provides three built-in global planners [1]:

- NavFN Planner: Implements a wavefront Dijkstra algorithm
- Smac Planner: Implements three A^* based planning algorithms
- Theta Star Planner: Implements the $Theta^*$ planner

As seen in the Nav2 architecture, once the global path has been computed, it is passed to the controller server, which performs the local path planning.

Local Planner

This section is largely based on [17]. It introduces the concept of local planners. These planners are a central component of this work, as the maps generated by the evolutionary algorithm should enable meaningful comparisons between them.

The local planner is utilized during the navigation process. Its primary objective is to follow the global path provided by the planner server while taking into account the robot's kinematics and dynamic constraints. These include, for example, the current velocity, acceleration limits, and the distance to near obstacles.

For this, it uses the local costmap for information about the local surroundings of the robot. There are a lot of parameters and constraints that can be considered, such as collision avoidance, smoothness of the path, or other self-designed constraints. For this purpose, the local planner utilizes the local costmap to obtain information about the robot's immediate surroundings. A variety of parameters and constraints can be taken into account, including collision avoidance and other user-defined constraints.

Depending on the application, controllers may implement different navigation strategies. These approaches can be categorized into three main groups:

- Reactive
- Predictive
- Geometric and Control-Law

In the following, local planners will be referred as *controllers*, as this is the common terminology in ROS 2/Nav2 and because the local planner generates commands that are sent directly to the robot's motor. Although Nav2 provides several built-in controllers [1], three controllers are considered in this work and used for comparison, each representing a different navigation strategy:

- Dynamic Window Based (Reactive)
- Model Predictive Path Integral (Predictive)
- Regulated Pure Pursuit (Geometric and Control Law)

Dynamic Window Based Controller

The Dynamic Window Based (DWB) controller is an implementation of the Dynamic Window Approach (DWA) [19] within the Nav2 framework. As the name suggests, the controller determines the next velocity command by evaluating candidate motions within a dynamically constrained velocity space. To do this, multiple velocity commands are sampled within the robot's control space. To estimate the resulting motion of the robot, a trajectory is forward simulated for each sampled command. These trajectories are then evaluated using a set of metrics such as progress toward the goal, alignment with

the global path, and distance to nearby obstacles [20]. Finally, the velocity command associated with the lowest overall cost is selected and executed by the robot.

Model Predictive Path Integral Controller

The Model Predictive Path Integral (MPPI) controller [1, 21] estimates an optimal trajectory by iteratively sampling and evaluating candidate trajectories. Multiple control sequences are forward-simulated, which are altered by Gaussian Noise. This results in a set of trajectories that are then evaluated using a predefined cost function. Using softmax weighting, the resulting cost values are used to compute an improved control sequence. This procedure is repeated n times. The resulting control sequence is serves as the initial input for the next time step.

Regulated Pure Pursuit Controller

The Pure Pursuit [22] controller operates by searching a target point on the global path that lies within a predefined *lookahead distance*. Based on the current pose of the robot and the selected point, an arc is constructed that connects both positions. The length of this arc corresponds to the lookahead distance. If there are multiple points within this range, the point closest to the robot along the path is chosen. Then, the target point is transformed into the robot's local coordinate frame. This enables the computation of the required curvature to reach this point. This is used to determine the angle which should be used for steering. This is given to the motor to update the motion of the robot.

The Regulated Pure Pursuit (RPP) controller [23] is an extension of this approach. It adjusts the linear velocities depending on the curvature of the path, thereby reducing high velocities in sharp corners. Furthermore, it incorporates built-in heuristics to slow down when obstacles are in close proximity to the robot, in order to reduce the risk of collisions.

2.2 Evolutionary Algorithms

Evolutionary algorithms are a class of metaheuristics. They are applied to a given problem in order to find a solution of high quality. It should be noted that there is no guarantee that the optimal solution will be found. The concept of evolutionary algorithms is inspired by biological evolution and translates its principles to computational problem-solving [24].

An evolutionary algorithm typically consists of the following components, largely based on the work of [24]:

Encoding

The encoding describes the internal representation of a candidate solution (referred to as *individual*). Choosing an appropriate encoding is crucial both for the design of the following operations and for algorithm's ability to effectively solve the problem.

Sampling

Sampling refers to the generation of an initial set of individuals, known as the *population*. The number of individuals in this set, called the *population size*, has a high impact on the performance of the evolutionary algorithm as larger population sizes increase diversity and the exploration of the search-space, while smaller populations can speed up convergence but increase the risk of early stagnation.

Fitness Function

The fitness function evaluates an individual and returns a fitness value with respect to the given problem. The resulting fitness value indicates how effectively the individual solves the given problem. Depending on the optimization goal, a higher fitness value may correspond to a better solution in the case of maximization, whereas a lower fitness value may indicate a better solution in the case of minimization.

Termination

The termination condition determines when the evolutionary algorithm should stop. Common criteria include:

- Reaching a maximum number of generations
- Reaching a maximum number of fitness evaluations
- Achieving a predefined target fitness value

Parent Selection

To generate new solutions, individuals must be chosen to create offspring. The selection of these individuals favors individuals with higher fitness, since they are likely to inherit attributes that lead to good solutions.

Typical selection methods include:

- **Roulette wheel selection**: Individuals are selected probabilistically in proportion to their fitness, giving individuals with better fitness a greater chance of being chosen.

- **Rank-based selection**: Individuals are ranked according to fitness, and selection probabilities are assigned based on rank rather than absolute fitness.
- **Tournament selection**: A small group of individuals is chosen randomly, and the fittest individual in this group is selected as a parent.

Crossover

Crossover is the operator applied to the selected parents to generate offspring. It is applied with a certain probability p_c During crossover, parts of the parents' encoding are exchanged, creating new individuals that combine traits from all parents. This operator promotes diversity and exploration of the search space.

Common Crossover methods include:

- **One-point crossover**: A single crossover point is selected, and the segments after this point are swapped between parents.
- **Two-point crossover**: Two crossover points are selected, and the segment between them is exchanged between parents.
- **n-point crossover**: Multiple crossover points are used, with alternating segments exchanged between parents.
- **Shuffle crossover**: The segments are randomly shuffled before applying crossover and then reshuffled back to their original order after recombination.

Mutation

Mutation is applied to newly generated offspring with a certain probability p_m. It introduces random modifications to the encoding, typically resulting in small changes in the solution space. Mutation helps preserve diversity within the population and lowers the risk of premature convergences to local optima.

Environmental Selection

After offspring are generated, the population temporarily exceeds its original size. Environmental selection reduces the population back to its predefined size by choosing which individuals will survive into the next generation. This process often employs mechanisms similar to parent selection, also favoring individuals with higher fitness.

Exploration/Exploitation

Exploration describes the algorithm's ability to search new and diverse regions of the solution space. Exploitation refers to refining and improving already promising solutions.

A good evolutionary algorithm balances exploration and exploitation in order to efficiently search the solution space while steadily improving solution quality.

A visualization of this cycle can be seen in Figure 2.3.

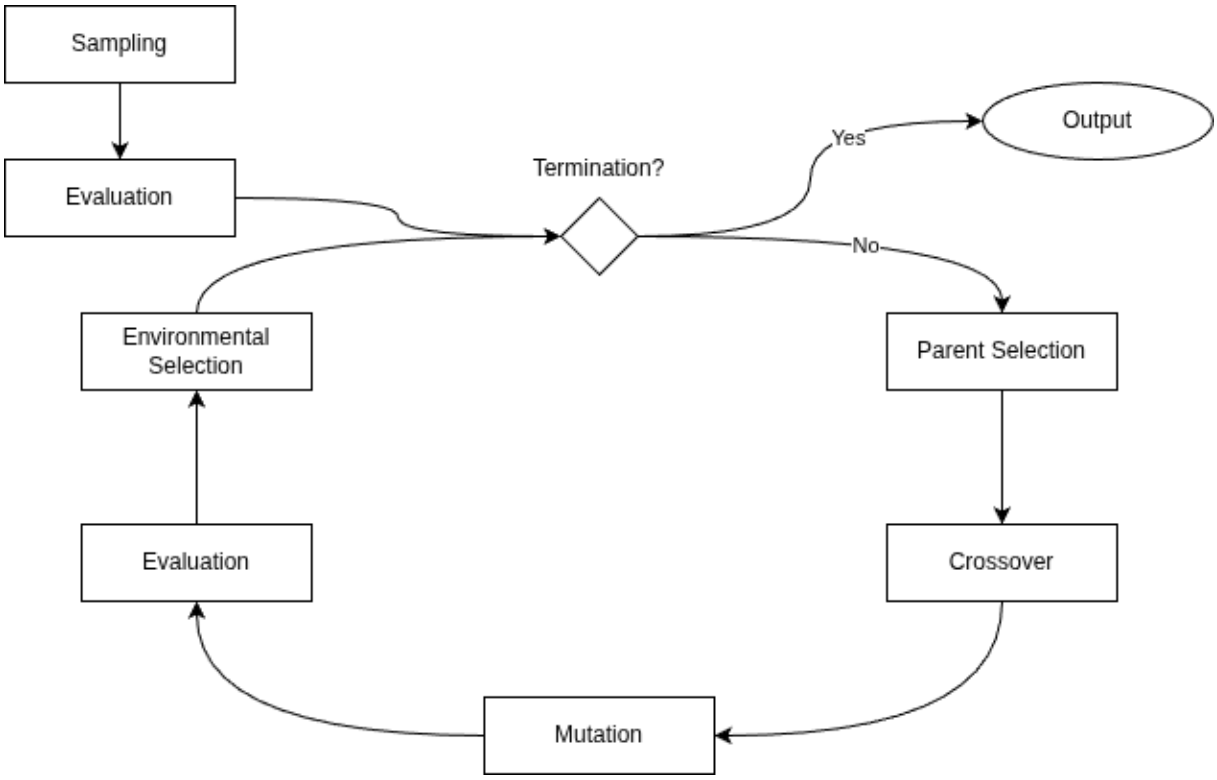

Figure 2.3: Cycle of an evolutionary algorithm

2.2.1 Pymoo

Pymoo is a Python framework designed for the solving of optimization problems using metaheuristic algorithms. It offers a modular architecture that allows the flexible definition of optimization problems. Additionally, it provides a wide range of algorithms and supports configurable genetic operators [9]. In this study, the Genetic Algorithm (GA) implementation within Pymoo is employed with custom-defined operators. The combination of Pymoo's flexibility and the robustness of its well-tested algorithms makes it suitable for both research purposes and practical applications.

3 Related Work

This section of the thesis covers related work of procedural content generation, the usage of evolutionary optimization in the field of robotics and finally comparisons of local trajectory planners. Afterwards, the research gap which this thesis should cover is presented.

3.1 Procedural Content Generation using EAs

Procedural content generation is a way to create primarily game content automatically, including assets, levels, and maps [25]. It is done to reduce manual development effort and to generate large amounts of content.

In the work of [26] the authors used a multiobjective evolutionary algorithm to procedurally generate complete and playable maps. They applied a search-based approach to create maps for an imaginary strategy game and for the game *StarCraft*. They defined multiple, partially conflicting fitness functions designed to capture perceived map quality. For the evolutionary optimization they used an SMS-EMOA as the selection mechanism to evolve grid-based maps. The result of the experiments showed a positive correlation between the optimized criteria and the perceived map quality. They conducted a user study with experienced StarCraft players, who generally provided positive feedback but noted that the maps did not match the quality of professional designed maps. Overall, the study demonstrates that complete and playable maps can be generated using evolutionary optimization.

Another study used a hybrid approach of a Genetic Algorithm and a Cellular Automaton to generate 2D mazes [27]. Instead of evolving complete maps directly, they evolved the rules of the cellular automaton, which in turn generated the maps. In this case, the approach was designed to learn and reproduce structural properties of the levels rather than generating full levels outright. Experimental results showed that this approach can generate maps exhibiting the desired structural attributes.

In the paper [28] the authors developed a tool for procedural generation of dungeon environments. For this they used a two population (FI-2Pop) genetic algorithm evolving

feasible and infeasible individuals in parallel. Similar to the previous paper, the system incorporated structural patterns in the level design process. The fitness of each individual is evaluated based on predefined structural patterns. Consistent with the findings of [26], the results showed a positive correlation between the target pattern ratios and actual pattern occurrence. Overall, the approach demonstrates that the system can generate diverse, playable maps, although precise control over pattern occurrences remain limited.

For readers that are interested in getting a broader overview of PCG techniques and applications, Togelius et al. (2013) [25] provide a comprehensive survey on procedural map generation and evolutionary methods for gameplay-aware content creation. Considering the results of the previously presented works, using evolutionary optimization to generate maps for robot navigation seems to be a promising approach.

3.2 Usage of Evolutionary Optimization in ROS 2

When considering the use of evolutionary optimization in ROS 2 or with autonomous mobile robots (AMRs), most existing work focus on optimizing navigation during execution. For example in [29], the authors applied metaheuristic path planning algorithms to optimize the execution time and path length of the global planner. For this purpose, the environments had to be manually constructed. In contrast to the approach presented in this thesis, their work focuses on optimizing the performance of global planners rather than generating environments automatically.

Another example of evolutionary optimization applied to path planning is presented in [30]. In this work, the author employs the evolutionary algorithm NSGA-II to address a multi-objective path planning problem for mobile robots in static environments. The algorithm simultaneously optimizes several objectives, including path length, safety with respect to obstacles and path smoothness. By generating a set of non-dominated solutions, the method provides different trade-offs between these objectives, from which an appropriate path can be selected. However, similar to the previously mentioned work, the focus lies on improving the quality of planned paths rather than on generating environments or benchmark scenarios for evaluating navigation algorithms.

3.3 Comparison of Local Trajectory Planners

Many studies have evaluated the performance of different controllers in ROS-based environments. These works compare controller such as DWB, TEB, EB, RPP, MPPI or RSC using different experimental setups or evaluation metrics [6, 7, 8, 31, 32].

The controllers were tested in various environments, including waypoint navigation tasks, narrow passages, warehouse-like layouts, and dynamic scenarios with moving obstacles or pedestrians. For example, [6] evaluates DWA, TEB and EB in both simulated and real-world environments for waypoint following. In [32], the author conducted experiments with narrow passages, while in [7] the controller performance was compared in a warehouse-like environment. For these comparisons, various evaluation metrics are considered in the literature. These include travel time, trajectory length, distance to obstacles, path smoothness, goal accuracy, computational resource consumption and the frequency of recovery behaviors.

Although these studies provide valuable insights into controller performance, they differ in terms of environments, controllers, and evaluation metrics. As a result, the comparability of these studies remains limited. To address this issue, existing work proposes standardized benchmarking approaches. For example [33] introduces a Mobile Robot Planning Benchmark that provides predefined simulation scenarios and evaluation metrics for systematic controller comparison. Different types of environments, including large-scale, partially unknown, and dynamic scenarios are evaluated in terms of safety, efficiency and trajectory smoothness.

3.4 Research Gap

To our knowledge, existing approaches for benchmarking controllers mostly rely on manually designed maps. While these benchmarks enable controllable and reproducible experiments, they offer only a limited number of navigation situations and may contain bias in the manually created maps. This limitation motivates the development of automated approaches to generate benchmark environments. Automatically generated navigation environments enable systematic evaluation of controllers across a wide range of scenarios. Since procedural content generation has proven to be a promising approach, in this work, evolutionary optimization in the form of a genetic algorithm is used to generate sets of environments to compare controller performance across diverse navigation environments.

4 Methodology

This chapter presents the proposed methodology. It first formalizes the design of the evolutionary algorithm, including the sampling strategy and the crossover and mutation operators. The two fitness functions used for evaluating individuals are then introduced. Finally supporting components such as map generation and navigation utilities are described, and the overall simulation workflow is explained.

4.1 Evolutionary Algorithm

In the following, two configurations of the evolutionary algorithm are presented. Both variants differ only in the definition of the fitness function. All other components and operators remain identical and are described in this section.

4.1.1 Encoding

Each individual represents a set of maps and is encoded as a Boolean tensor of size $n \times m \times m$, where the first dimension corresponds to the number of maps and the remaining dimensions define the width and height of each map. Each cell is binary, indicating either free space or an obstacle. This discrete representation allows efficient manipulation through evolutionary operators and can be directly transformed into the grid representations underlying local and global costmaps for robotic navigation systems.

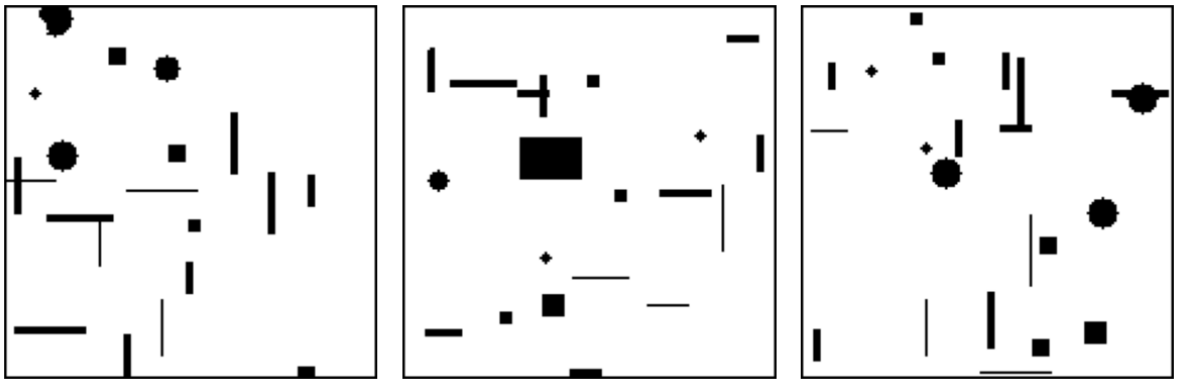

Figure 4.1: Example individual with $n = 3, m = 150$

4.1.2 Sampling - Shape

The initial population is generated using a shape-based sampling strategy. Each map is populated with a number of randomly placed obstacles. For every obstacle, a primitive geometric shape is selected from a predefined set. For each shape, a random position and size are sampled. Wall segments may additionally have a randomly assigned orientation and thickness. For wall segments, orientation (horizontal or vertical) and thickness (between 1 and 3 cells) are also randomly determined. After placing the obstacles, two predefined positions representing the start and goal locations are cleared to ensure they are free of obstacles. Furthermore, the map boundaries are filled with obstacles to prevent the robot from leaving the environment to bypass obstacles.

4.1.3 Parent Selection

Parent selection is performed using the standard selection mechanism provided by Pymoo [9], specifically a binary tournament strategy. In each tournament, two individuals are randomly chosen and compared based on their fitness values, with the individual with the better fitness selected as a parent. This method ensures sufficient selection pressure while preserving diversity in the population.

4.1.4 Crossover - Pool Based

Initially, the crossover was implemented in a position-dependent manner. For each offspring, maps were selected from either parent with equal probability. However, this

approach assumes a consistent ordering of maps within individuals. Since the objective is to generate sets of maps with gradually increasing differences in the behavior of local trajectory planners, this assumption does not hold. Since the increasing differences are not tied to a fixed position within the individual, the ordering of maps is irrelevant. To account for this, a position-independent set-based crossover was introduced. Instead of exchanging maps at fixed positions, the maps of both parents are combined into a shared pool. From this pool, n maps are randomly selected to form the first offspring, while the remaining maps form the second offspring.

4.1.5 Mutation - Shape

Mutation operates on detected obstacle structures within each map. Using connected-component labeling from SciPy [34], individual shapes are identified. Prior to mutation, the map borders are temporarily removed to preserve the outer boundary constraints. After mutation, both the borders and the predefined start and goal regions are restored.

With probability p_c, a mutation step is applied to an individual. For each map k shapes are selected for modification. Four mutation operations are applied:

- Move: The selected obstacle is translated by a random offset.
- Scale: The obstacle is enlarged or reduced. During this process, discretization effects may cause geometric transformations (e.g.,squares becoming circular-like).
- Remove: The obstacle is deleted.
- Add: A new obstacle is generated using the same procedure as in the sampling step.

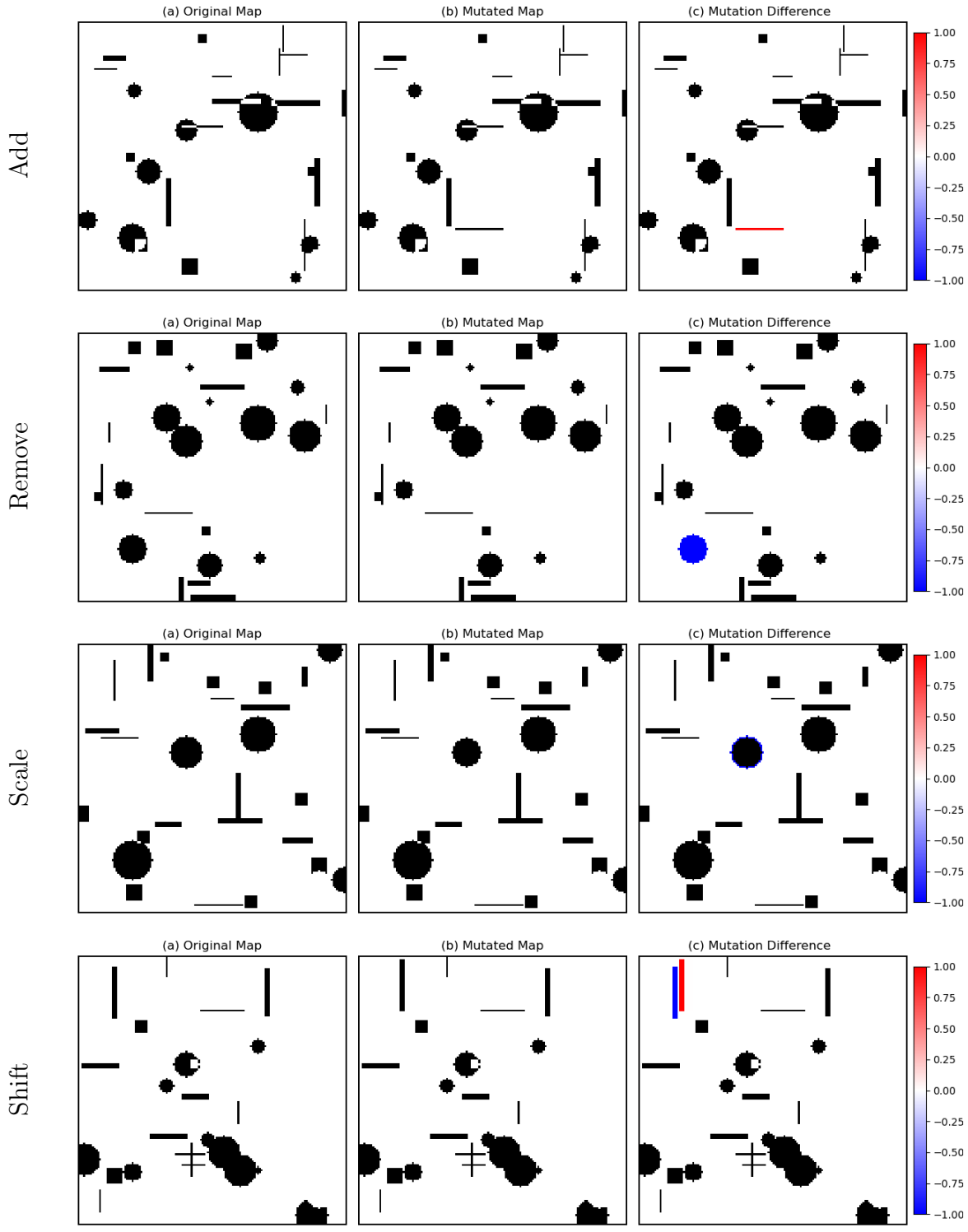

Figure 4.2: Visualization of the four mutation operators

4.1.6 Environmental Selection

For environmental selection, the standard survival mechanism provided by Pymoo is used. Individuals are ranked according to their fitness values, and the best-performing individuals are retained for the next generation. This elitist strategy promotes convergence toward high-quality solutions.

4.1.7 Duplicate Elimination

To maintain diversity and prevent premature convergence, duplicate individuals are removed from the population. Two individuals are considered duplicates if their Boolean map representations are identical.

4.1.8 Gini - Fitness function

The Gini coefficient is a measure of inequality [35], indicating how unevenly a resource is distributed among a set of agents. Its value G lies in the closed interval $[0, 1]$, where $G = 0$ corresponds to perfect equality, such that all agents possess identical amounts of the resource and $G = 1$ indicating maximal inequality, in which a single agent holds the entire resource while all others hold none. Mathematically, the Gini coefficient can be interpreted in terms of the **Lorenz curve** $L(x)$, which represents the cumulative proportion of the resource held by the bottom x fraction of the population of agents. Letting the line of perfect equality represent the idealized uniform distribution of the resource, the Gini coefficient corresponds to the normalized area between the Lorenz curve and this line of perfect equality. Formally, it is expressed as:

$$G = 1 - 2 \int_0^1 L(x)\, dx, \tag{4.1}$$

where $L(x)$ denotes the Lorenz curve as a function of the cumulative population share x [35].

A schematic visualization of this relationship is provided in Figure 4.3, illustrating the geometric interpretation of the coefficient as the area between the Lorenz curve and the line of equality.

In the context of our study, inequality in the distribution of travel times across the different controllers on a map can be interpreted as a performance disparity. Consequently, we adopt the Gini coefficient as a component of a fitness function.

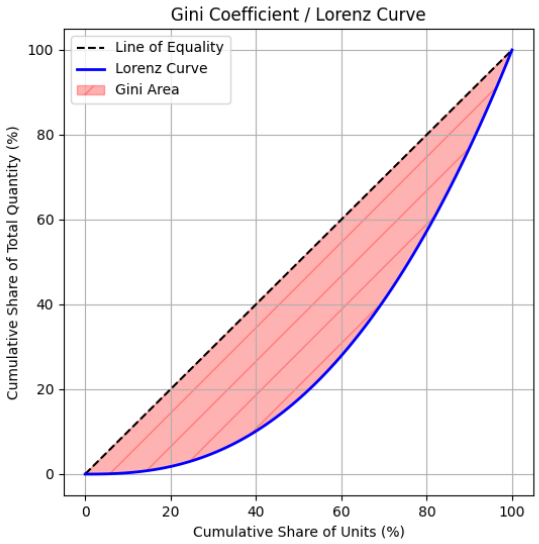

Figure 4.3: Plot of the Lorenz curve and Gini Area

In the context of this thesis, the Gini coefficient serves as a metric to evaluate inequality in performance durations among controllers evaluating a given map. A coefficient of 0 indicates that all planners perform equally well on a given map, while a value close to 1 reflects strong disparity, meaning at least one planner requires significantly more time than the others.

Consequently, we adopt the Gini coefficient as a component of a fitness function to generate maps that produce varying levels of performance disparity across different controllers.

We define a function *ginis()* which takes n time values as an input and then calculates the gini values of these. We adopt a rank-based formula in a modified form, following Equation (10) in [36]:

$$g = \frac{2\sum_{i=1}^{n} i\, t_i}{n\sum_{i=1}^{n} t_i} - \frac{n+1}{n} \tag{4.2}$$

The resulting implementation is presented in Algorithm 1.

Once the Gini coefficients have been computed, a corresponding fitness value must be determined for the given combination of coefficients. To this end, two constraints are considered. First, the distance between the sorted Gini values should be as uniform as possible. Second, the overall spread of the sorted values should be maximized. Those constraints are intended to assign high fitness scores to individuals whose maps produce

an evenly increasing disparity in controller performance while simultaneously exhibiting a clearly distinguishable difference in performance between controllers.

The computation of this fitness value is shown in Algorithm 2.

Algorithm 1: Compute Gini coefficient from a list of time values

Input: times: list of n values $[t_1, t_2, \ldots, t_n]$
Output: Gini coefficient g
times $\leftarrow$ sort(times);
n $\leftarrow$ length(times);
for $i \leftarrow 1$ **to** n **do**
  index[i] $\leftarrow i$;
end
numerator $\leftarrow 2 \cdot \sum_{i=1}^{n} index[i] \cdot times[i]$;
denominator $\leftarrow n \cdot \sum_{i=1}^{n} times[i]$;
g $\leftarrow$ numerator / denominator $-(n+1)/n$;
return g;

Algorithm 2: Compute the fitness value for a list of Gini coefficients

Input: ginis: list of Gini coefficients $[g_1, g_2, \ldots, g_n]$, ϵ (small threshold, default $1e-8$), λ (weight for unevenness, default 1.0)
Output: fitness value f
ginis $\leftarrow$ sort(ginis);
if *length(ginis)* < 2 **then**
  return -10^3;
end
deltas $\leftarrow$ differences between consecutive ginis;
if *any deltas* $\leq \epsilon$ **then**
  return -10^3;
end
spread $\leftarrow$ ginis[-1] $-$ ginis[0];
unevenness $\leftarrow$ variance(deltas);
f $\leftarrow$ spread $-$ $\lambda\cdot$ unevenness;
return f;

4.1.9 Cluster - Fitness function

For the second fitness function, we consider an idealized distribution of controller execution times for n controllers on a generated map, as illustrated in Figure 4.4 with $n = 3$ as an example. The goal is to generate a set of n maps that exhibit progressively increasing disparity in controller performance.

The execution times can therefore be interpreted as one-dimensional data points, as illustrated in Figure 4.5. For visualization purposes, the figure shows the case of three controllers. Based on this representation, controllers with similar execution times should form groups, whereas sufficiently distant values should belong to different groups or clusters.

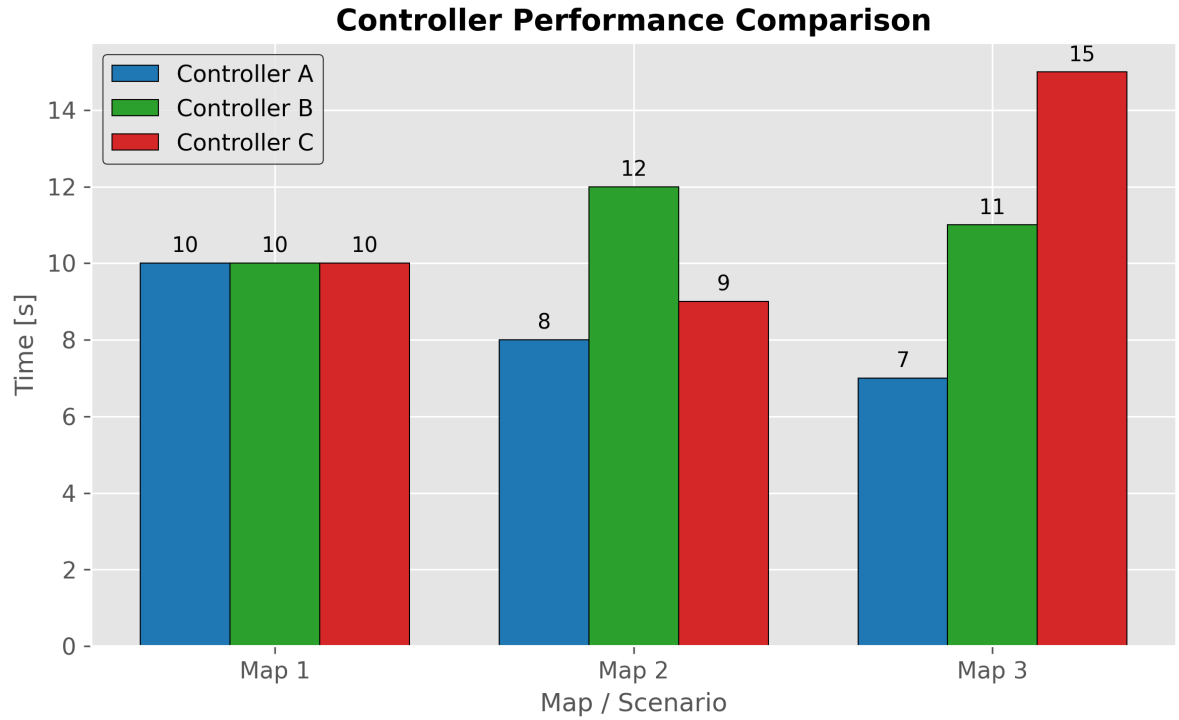

Figure 4.4: Ideal distribution of an individual's times

4.1.10 Cluster Score

Before estimating the number of cluster, it is important to note that the execution times alone do not directly determine whether values belong to the same group or to different groups. Without an appropriate scale, it is impossible to distinguish situations in which

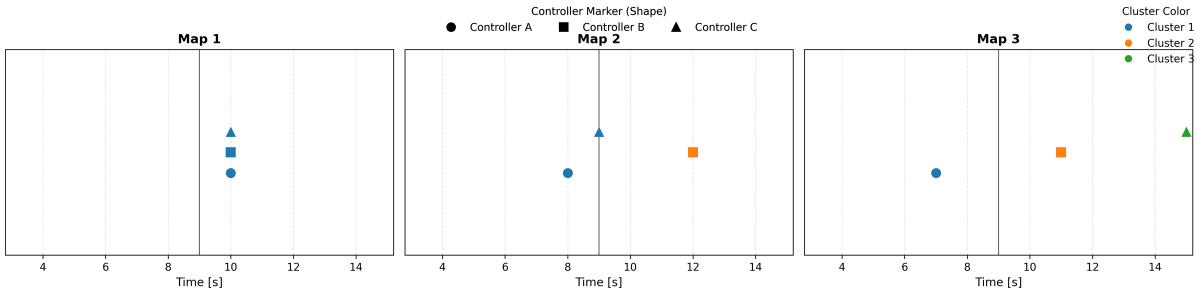

Figure 4.5: Ideal assignment to clusters depending on driving times

all controllers perform similarly and situations in which all execution times differ. To address this issue, a reference scale is introduced that defines what constitutes a small or large difference between execution times. This scale is used to normalize the pairwise distances between the measured times.

The idea of the score function is illustrated in Algorithm 3. First, all pairwise distances between the execution times of the controllers are computed. The mean of these distances is then used as a measure of overall spread of the values. Based on this mean distance, an estimate of the number of clusters is computed as

$$k = 1 + (meandist/scale) \tag{4.3}$$

where *scale* denotes the reference value used to normalize the distances and k denotes the estimated number of clusters. This estimated cluster count is subsequently used to derive the fitness value for the corresponding set of controller execution times. Now that we got the amount of clusters of the n time values we can now focus on calculating a fitness for them.

Algorithm 3: Compute cluster score from n time values

Input: times: list of n values $[t_1, t_2, ..., t_n]$, scale
Output: cluster score c in $[1, n]$

```
distances ← empty list;
for i from 0 to n − 2 do
    for j from i + 1 to n − 1 do
        distances.append(|times[i] - times[j]|);
    end
end
mean_dist ← sum(distances) / length(distances);
c ← 1+ (mean_dist / scale);
if c < 1 then
    c ← 1;
end
else if c > n then
    c ← n;
end
```

4.1.11 Pattern Score

The next step is to assign a fitness value to the specific set of clusters obtained from each map. Ideally, the number of clusters across the maps should increase by one for each map when sorted. Therefore, we define an ideal pattern $[1, 2, ..., n]$.

To evaluate an individual, we first sort the obtained cluster values and compare them to this ideal pattern. For each position i, we compute the difference between the observed number of clusters and the corresponding value in the pattern. The squared errors are summed and the square root is taken to obtain the Euclidian distance between the observed cluster distribution and the ideal pattern.

Finally, this distance is normalized by dividing it by the maximum possible distance to the pattern. The resulting fitness score is defined as

$$score = 1 - \frac{distance}{max\_distance}, \tag{4.4}$$

which yields a value between 0 and 1, where values closer to 1 indicate a cluster distribution that more closely matches the ideal pattern.

However, additional cases must be considered. During the evaluation of individuals it is possible that results are not obtained for all n maps. This may occur if a generated map is

infeasible for the controllers or if issues arise in ROS-based communication. Consequently, the number of cluster values returned may be smaller than n.

This situation occurs more frequently during the early iterations of the evolutionary algorithm, as randomly generated maps are more likely to be infeasible. Therefore, special handling is required for individuals that produce only two or fewer cluster values. If only two cluster values are available, the fitness is based on the distance between them. The ideal distance for this is 1. The closer the observed distance is to this value, the better the resulting fitness. Distances larger than 1 are treated equally, as they may simply indicate that results from additional maps are missing.

If fewer than two cluster values are available, no meaningful comparison can be made and the fitness is set to 0.

Algorithm 4: Cluster Pattern Fitness Evaluation

Input: values: list of numerical values

Output: score between 0 and 1

Function `pattern_score(`*values*`):`

```
n ← LENGTH(values);
pattern ← [1,2,...,n];
values_sorted ← SORT(values);
distance ← SQRT( SUM over i of (values_sorted[i] - pattern[i])^2 );
min_val ← MIN(pattern);
max_val ← MAX(pattern);
max_distance ← SQRT( SUM over i of (max_val - pattern[i])^2 );
score ← 1 - distance / max_distance;
RETURN MAX(score, 0);
```

Function `partial_pattern_score(`*values*`):`

```
n ← LENGTH(values);
if n = 0 OR n = 1 then
    RETURN 0.0;
end
else if n = 2 then
    v1 ← values[0];
    v2 ← values[1];
    dist ← ABS(v1 - v2);
    target_dist ← 1.0;
    score ← dist / target_dist;
    RETURN MIN(MAX(score, 0), 1);
end
else if n > 2 then
    RETURN pattern_score(values);
end
```

4.1.12 Fitness Composition

In addition to the pattern-based fitness functions, further constraints must be considered in order to evaluate the quality of an individual. Therefore, the final fitness value is composed of several penalty terms. The value returned by the pattern-based fitness functions is referred to as the pattern cost. Since the optimization problem is formulated

as a minimization problem, the pattern cost produced by both fitness functions is negated before being used in the final fitness calculation.

As the algorithm aims to generate map sets in which all controllers are able to successfully navigate, a success ratio constraint is introduced. This constraint penalizes individuals containing maps where one or more controllers fail to produce a valid plan, as well as individuals containing maps that are generally not solvable. In addition, the composed fitness value considers simulation runs that terminate prematurely due to errors and are subsequently treated as timeouts. These timeouts are explicitly counted and incorporated as penalty terms in the final fitness value in order to discourage unreliable behavior. The final fitness value is computed as

$$f = C_b(1 - s) + C_t\, t + w_p\, c_p\, s \tag{4.5}$$

where the first term penalizes unsuccessful planning attempts, the second term penalizes timeouts occurring during the simulation, and the third term incorporates the pattern cost produced by the fitness function while weighting it by the success ratio. This formulation ensures that individuals are not only rewarded for generating map sets with desirable pattern characteristics, but also for producing maps that can be reliably solved by the evaluated controllers.

4.2 Map Generation

The representation used within the evolutionary algorithm is not directly compatible with the requirements of the ROS-based simulation environment. As described in the *Encoding* subsection of the *Evolutionary Algorithm* section, each individual consists of a Boolean array of size $n \times m \times m$. These n layers represent separate maps, where each cell is encoded as either 0 or 1.

However, this format cannot be used directly to initialize a ROS simulation. Therefore, a conversion step is required to transform the internal representation into a set of files that are compatible with the simulation and navigation framework. For this purpose, a dedicated Python class named `mapconverter.py` was implemented. This class provides a method called `createMaps`, which takes a list of Boolean arrays as input and iteratively converts each individual into the required simulation assets.

During this process, the method automatically generates the following files:

- Occupancy grid images in `.pgm` format

- Corresponding metadata files in `.yaml`
- Robot pose configuration files (`map*poses.yaml`)
- Gazebo simulation world files (`.world`)
- Simulation model configuration files (`model.config`)
- Simulation model description files (`model.sdf`)
- 3D mesh representations (`.dae`)

This conversion pipeline enables a transition from the abstract individual encoding used in the evolutionary algorithm to a fully functional simulation environment in Gazebo. An example of the transformation from the Boolean array representation to the generated simulation map is shown in Figure 4.6.

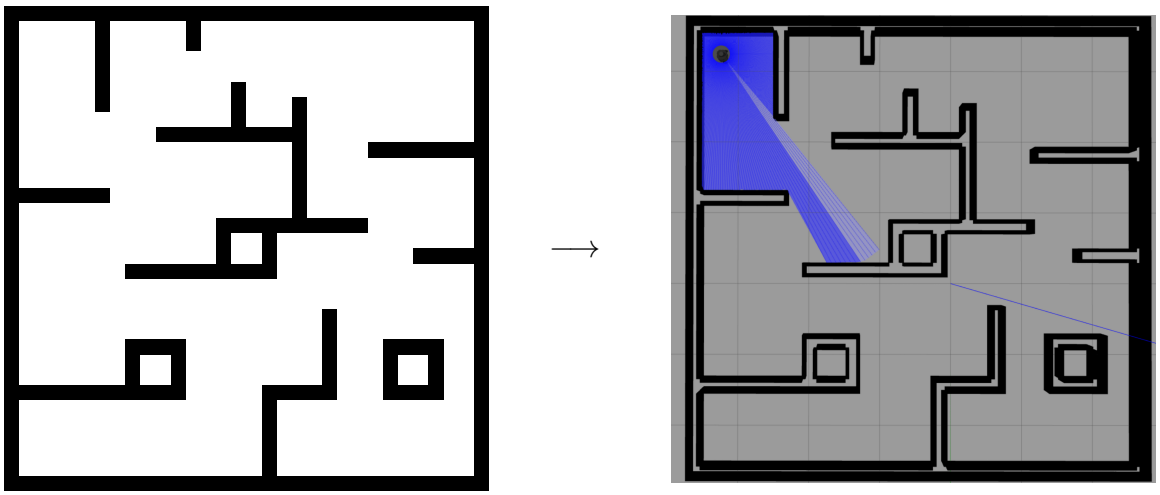

Figure 4.6: Transformation of a Boolean array to a simulated Gazebo map

4.3 Simulation Control Nodes

To evaluate the generated maps and navigation behavior within the simulation environment, a ROS 2 node was implemented to send navigation goals to the robot. This node, referred to as the *Goal Sender Node*, communicates with the navigation stack and triggers the execution of a navigation task.

The node uses the `NavigateToPose` action provided by the BT Navigator. It accepts the parameters x, y, and w, which define the target position and orientation of the robot. Additionally, a parameter *bt_xml_file* specifies the behavior tree of the controller that should

be used during navigation. When a goal is issued, the node sends the corresponding target pose to the navigation action server and monitors the execution of the navigation task while measuring the elapsed time. To ensure robustness during automated experiments, a timeout mechanism is implemented. If the robot fails to reach the goal within a predefined maximum runtime, the navigation task is automatically canceled. This prevents the evaluation pipeline from blocking due to navigation failures or controller deadlocks.

After the navigation task has completed, the node evaluates the returned action status and determines whether the goal was successfully reached. The method returns a tuple consisting of a Boolean value indicating success or failure, as well as the total execution time required for the navigation attempt. These values are later used by the evolutionary algorithm to compute the fitness of individuals.

To ensure reproducible navigation experiments within the simulated environment, the robot must be reset to a defined start configuration after each navigation attempt. For this purpose, a ROS 2 Node referred to as the *Set Position Node* was implemented.

The node resets the simulation state by removing the current robot instance from the Gazebo environment and spawning a new robot at a specified position and orientation. The desired pose is defined by the parameters x, y, and yaw. A configurable timeout parameter is included to prevent the reset procedure from blocking the evaluation pipeline in case of simulation errors. The reset procedure first removes the existing robot model using the Gazebo service `/delete_entity`. Afterwards, a new robot instance is spawned at the specified coordinates. To increase robustness, the spawning procedure includes retry attempts if the initial request fails. Once the robot has been successfully spawned, the node can optionally publish an `/initialpose` message to the localization system (AMCL) to ensure that the robot's pose estimate matches the newly defined start position. This reset mechanism guarantees identical initial conditions for subsequent navigation runs and therefore enables consistent automated evaluation during the evolutionary process.

4.4 Simulation

A complete evaluation cycle of the system components is performed as follows. The evolutionary algorithm evaluates the fitness of the population by invoking the fitness function for each individual. When the fitness function is called, the maps encoded in the individuals are first converted into simulation-ready files using the already introduced *createMaps* function. These generated maps are then passed to the function *simulate()*, which performs the navigation experiments in the ROS 2 simulation environment. The

simulate() function iterates over all n maps contained in the individual. For each map, a ROS 2 context is initialized and the simulation is started via the corresponding ROS launch file. Once the simulation environment is fully initialized, the system iterates over all defined controllers. For each controller, the corresponding behavior tree is sent to the Goal Sender node, which initiates the navigation process. During execution, each controller is tasked with navigating the robot from the lower-right to the upper-left corner of the map, starting from a fixed initial position. During execution the system records whether the goal was successfully reached as well as the execution time required for the navigation task. These results are stored in a dictionary structure for further processing. After each controller run, the robot position is reset using the Set Position Node to ensure identical conditions for subsequent controller evaluations. After all controllers have been evaluated on all maps, the collected performance data is returned to the fitness function. The recorded execution times and success indicators are then processed to compute the final fitness value. Depending on the selected fitness formulation, metrics such as runtime distribution, controller success rate, and the occurrence of timeouts are incorporated into the overall fitness. A simplified visualization of the simulation flow without retry or timeout mechanisms can be seen in Figure 4.7.

Sometimes the simulation may stall or stop to response. For this, additional error-handling mechanism were implemented. Each map evaluation was assigned a limited number of retry attempts. In cases where execution failed or a component became unresponsive, the corresponding processes were terminated and the simulation environment was restarted. If all retry attempts failed without successful completion, the respective evaluation was unsuccessful and penalized accordingly within the fitness calculation.

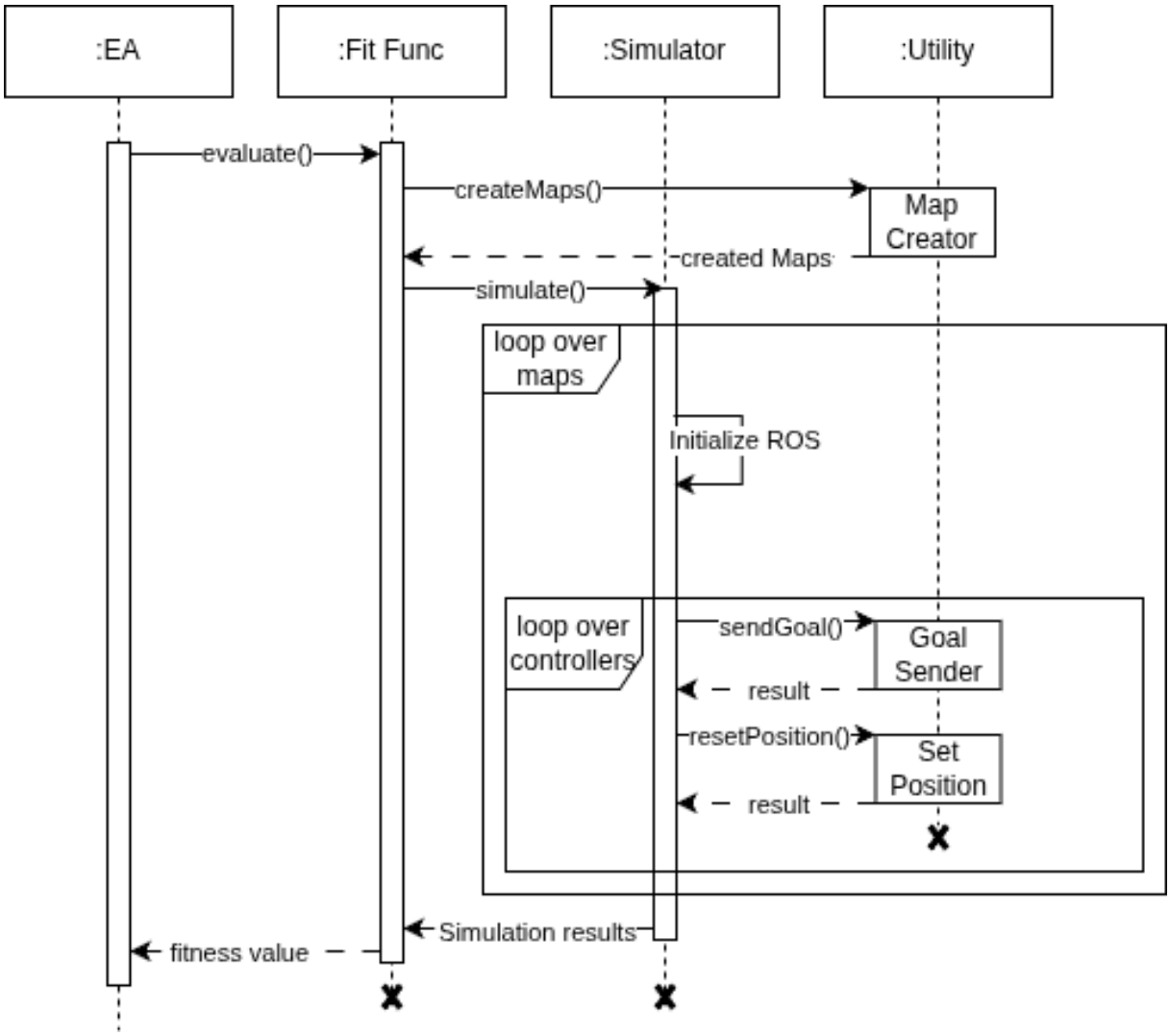

Figure 4.7: Sequence of simulation

4.4.1 Parallel Evaluation

The fitness evaluation relies on a physic-based simulation running in real time, which makes the evaluation of each individual computationally expensive. Assuming that each individual is evaluated on n maps using n planners, where each planner requires around t seconds per map and the simulation requires an additional startup time of c seconds, the evaluation time per individual can be approximated as

$$T_{\text{individual}} = t \cdot m \cdot n + n \cdot c \tag{4.6}$$

For large populations or a high number of evaluations, this results in substantial overall runtime. To address this issue, multiple individuals are evaluated concurrently (see RQ4).

However, running multiple ROS 2 and Gazebo simulations simultaneously introduces additional challenges. Because of ROS 2 Usage of Discovery, without proper isolation, concurrently running simulations may interfere with each other. To prevent this, each evaluation is assigned a unique *ROS_DOMAIN_ID*, ensuring separation between independent simulations. A similar issue arises for the Gazebo simulation, where sensor data streams from different instances may conflict. This is addressed by assigning a distinct *Gazebo Master URI* to each simulation instance. By isolating the simulations in this way, this approach allows multiple evaluations to be executed in parallel, significantly reducing the total evaluation time.

5 Experiments

This chapter outlines the experimental design, including the parameter configurations and the global planner and controllers that are employed. Subsequently, the results of the proposed approach are presented, followed by a benchmark evaluation of the resulting individuals.

5.1 Setup

All experiments were conducted within the Driving Swarm Infrastructure framework using ROS 2 Humble and Gazebo for simulation. The evolutionary algorithm was implemented using the Pymoo optimization framework. Experiments were executed on a system containing an AMD Ryzen 9 5900X CPU, an Nvidia RTX 4060 Ti GPU and 32 GB DDR4 RAM.

For the navigation, the built-in NavFN planner of Nav2 was used as a global planner and the evaluated controllers were the previously introduced DWB, MPPI and RPP controllers. The controller parameters were configured according to the example configurations provided in the official Nav2 documentation.

Sampling, crossover and mutation operators were identical across all experiments. The mutation probability p_m was set to 0.5 and the crossover probability p_c was set to 1.

For all operators provided by Pymoo, each run was initialized with a fixed random seed to ensure reproducibility of the evolutionary process. Consequently, the sequence of generated individuals and applied genetic operations is deterministic for a given seed.

Obstacle patterns were generated using three primitive shapes:

- Circle: radius between 2 and 10 cells
- Square: side length between 4 and 10 cells
- Wall: length between 10 and 30 cells

During sampling, the following probability distribution was used:

- Circle: $p = 0.2$
- Square: $p = 0.3$
- Wall: $p = 0.5$

These probabilities were chosen to bias the initial maps toward simpler structures. Walls represent the least complex obstacle primitive, while squares correspond to combinations of adjacent walls and circles introduce more complex geometries. This distribution therefore reduces the likelihood of overly complex maps during early generations.

During the experiments using the Gini-based fitness function, driving times of the controllers were recorded in order to determine an appropriate scale parameter for the cluster fitness function.

As a reference scale, the median of the observed distances between driving times was used, resulting in a value of 2.7069. The median was preferred over the mean because it is more robust to outliers caused by occasional ROS communication errors during simulation.

For the penalty formulation, the following parameters were used: a base cost $C_b = 100$, a pattern weight $w_p = 20$, and a timeout cost $C_t = 10$. A comparatively high base cost was selected to ensure that the evolutionary algorithm primarily prioritizes the generation of valid maps. Timeouts occurred frequently during early experiments and could not be fully eliminated. Therefore, a moderate timeout penalty was applied.

The pattern weight w_p determines the influence of the actual fitness relative to the structural penalties. A value of 20 was chosen to ensure that performance differences between controllers remain a relevant optimization while still enforcing map validity constraints.

Initially, three experimental series were conducted using the Gini fitness with population sizes of 20, 40, and 60 individuals to investigate the effect of different population sizes on the performance of the EA (see RQ3). Each configuration was executed for 24 runs with a maximum of 600 fitness evaluations. The purpose of these experiments was to determine a suitable population size for subsequent evaluations. After identifying the most appropriate population size and the scale parameter, the same experimental setup was applied to the Cluster fitness function.

While the evolutionary algorithm itself is deterministic for a given random seed, the ROS/Gazebo simulations were not seeded. Consequently, identical individuals evaluated in repeated simulation runs may produce slightly different results.

Table 5.1: Overview of experimental setups

Fitness Function	Population Size	Runs	Max Evaluations
Gini	20	24	600
Gini	40	24	600
Gini	60	24	600
Cluster	optimal size*	24	600

*Optimal population size determined from Gini experiments

5.2 Results

First, the results of the evolutionary algorithm using the Gini-based fitness function are analyzed for three different population sizes: 20, 40, 60. Each configuration was executed for 24 independent runs.

The fitness development over generations is shown in Figure 5.1. For population sizes of 20 and 40, the average minimum fitness improves considerably during the first generations, indicating that the evolutionary algorithm quickly identifies better individuals early in the search process. A similar trend can be observed for the population size of 60, although the improvement is slightly less pronounced compared to the other two configurations.

Across all three configurations, the curves representing the average fitness follow a similar overall shape. Furthermore, none of the runs appear to fully converge within the given number of generations. By the end of the evolutionary process, all configurations reach comparable levels of average minimum fitness. The configuration with a population size of 40 also shows a smaller variance in the fitness values.

To further analyze these observations, statistical metrics of the final fitness values across the 24 runs were computed.

Population Size	**Mean**	**Std**	**Median**
20	-8.57	1.20	-8.54
40	-8.77	0.85	-8.64
60	-8.59	1.39	-8.69

Table 5.2: Statistical summary of the final fitness values across 24 runs for different population sizes using the Gini fitness function.

The results indicate that the configuration with population size 40 achieves the lowest mean fitness value and the smallest standard deviation among the tested configurations.

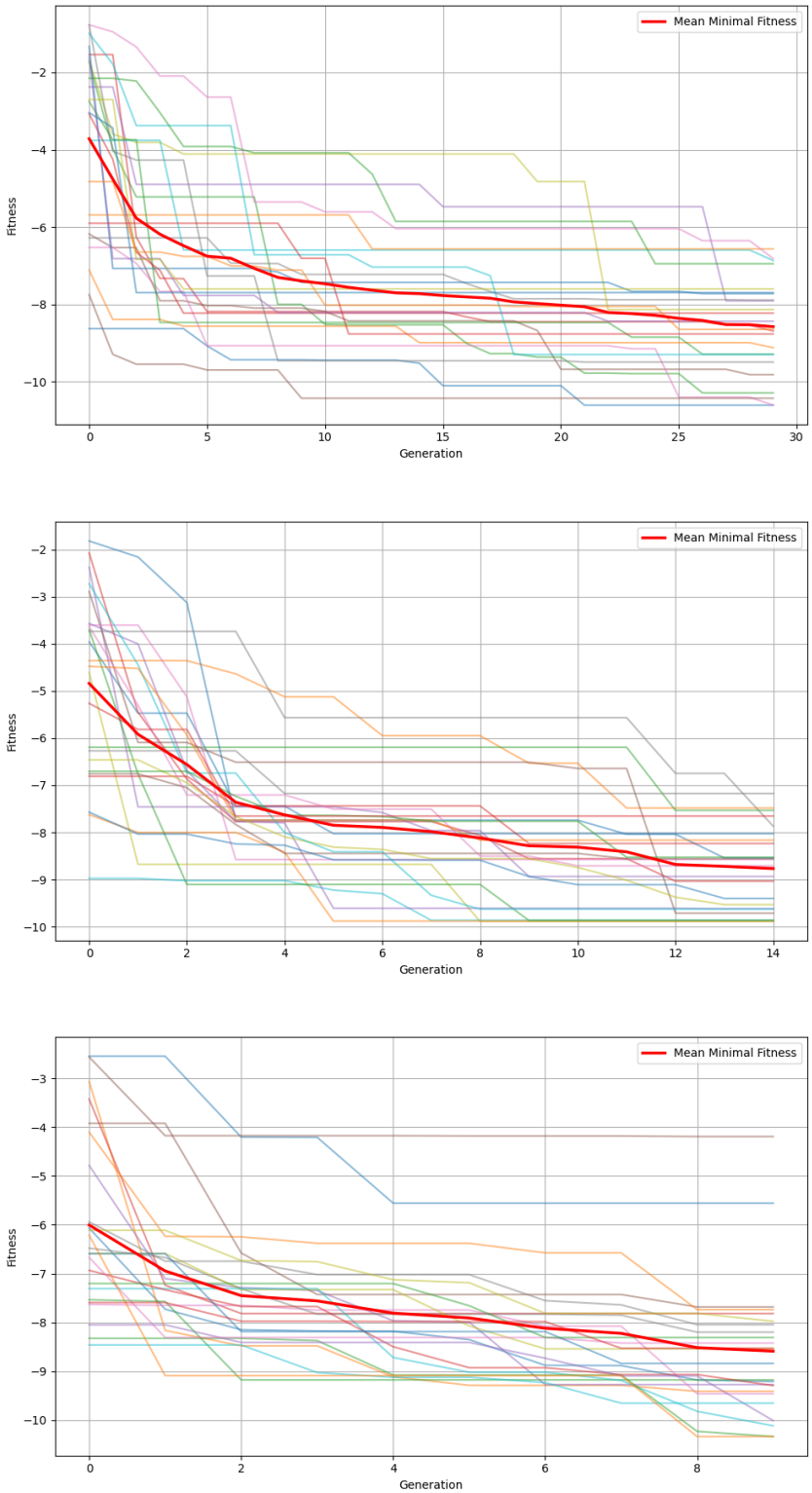

Figure 5.1: Minimal fitness per generation for the Gini fitness function with population sizes 20, 40, and 60.

This observation is also supported by the boxplot shown in 5.2, where the distribution of the final fitness values for population size 40 appears more compact compared to the other configurations.

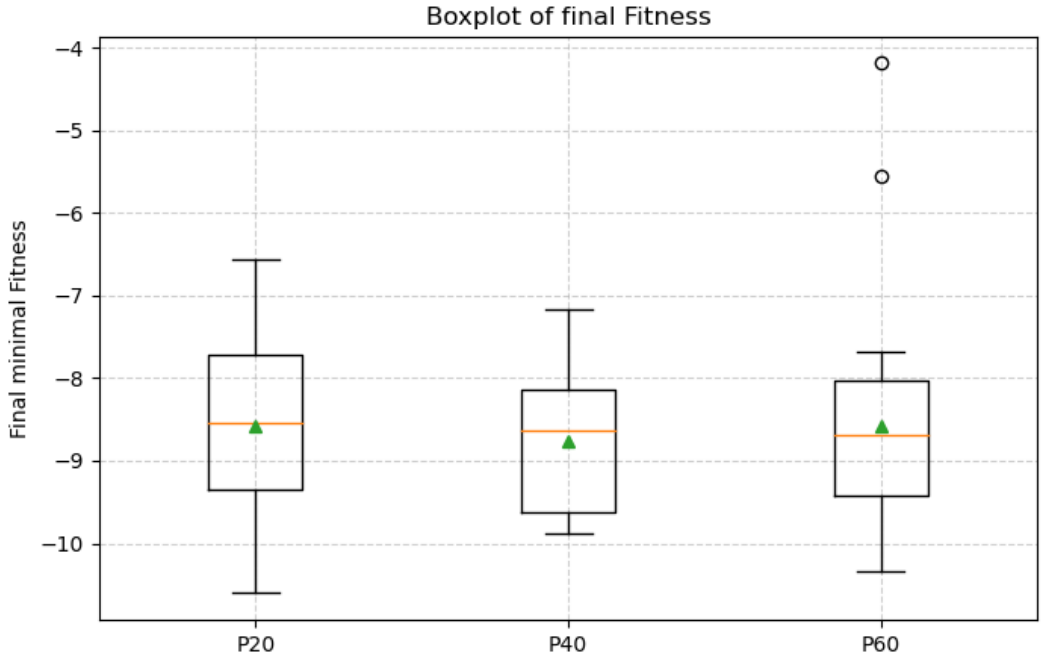

Figure 5.2: Box plot of the 3 Gini results

To assess whether the observed differences between the configurations are statistically significant, a Kruskal-Wallis test [37] was performed. The test yielded a statistic of 0.44 and a p-value of 0.80. Since the p-value is considerably higher than the commonly used significance threshold of 0.05, the null hypothesis H0 from RQ3 cannot be rejected. This indicates that there is no statistically significant difference between the tested population sizes. Nevertheless, the configuration with a population size of 40 was selected for the subsequent experiments using the cluster-based fitness function. This choice was motivated by its slightly better average fitness and the lower variance observed across runs.

5.2.1 Analysis - Gini Individual

Figure 5.3 presents the best individual obtained using the Gini-based fitness function with a population size of 40. Map 1 exhibits a highly non-uniform distribution of ob-

stacles. A dense cluster of objects is concentrated in the lower-left quadrant, resulting in a locally constrained region. In contrast, the area extending from lower-right toward the upper-right and further to the upper-left remains relatively sparse, forming a largely unobstructed corridor across the map Map 2 and Map 3 are nearly identical in their spatial configuration. The differences between them are minimal. Map 3 contains one fewer object and a single square-shaped obstacle is slightly repositioned compared to Map 2. Overall, both maps maintain a very similar obstacle distribution and structural layout.

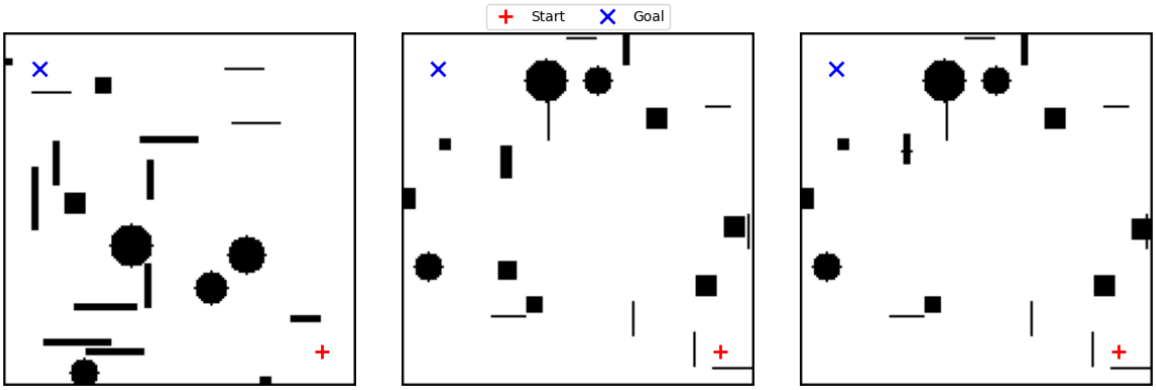

Figure 5.3: Visualization of the best individual using the Gini fitness function

5.3 Result Cluster

The fitness development for the cluster-based fitness function is shown in Figure 5.4. Similar to the Gini-Based fitness function, the evolutionary algorithm shows a strong improvement during the first generations. However, the improvement slows down towards the end of the optimization process. As with the previous experiment, the algorithm does not appear to fully converge within the given number of generations.

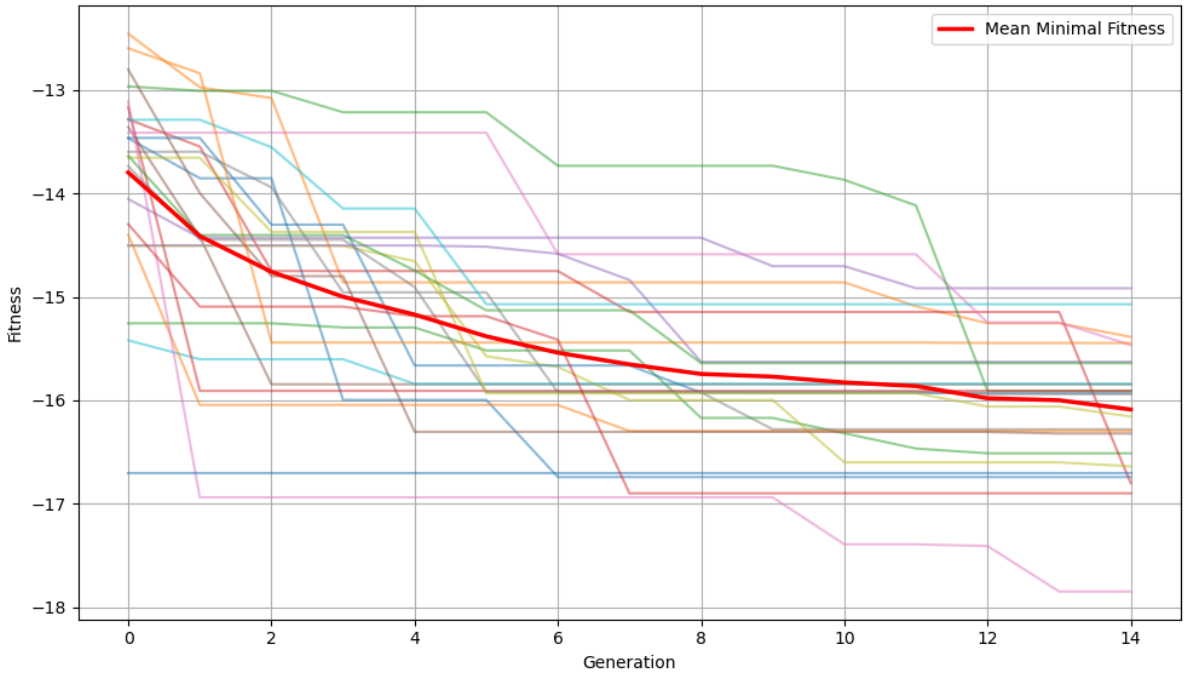

Figure 5.4: Minimal fitness per generation for the cluster fitness function with population size 40

5.3.1 Analysis - Cluster Individual

Figure 5.5 shows the best individual obtained from the cluster fitness function. In Map 1, the path from the lower-right to the upper-left is strongly shaped by a dense concentration of obstacles located slightly below the horizontal center. As a result, the traversable space forms an S-shaped corridor, requiring the controller to perform multiple directional changes along the route. In contrast, Map 2, despite containing a high number of obstacles, allows for a largely unobstructed path from the lower-left to the upper-right. The resulting trajectory is therefore comparatively simple, featuring only a short narrow passage at the beginning before opening into a more spacious and easily navigable area. Although Map 3 contains relatively few obstacles, their specific placement restricts the available paths. In particular, the lower-right quadrant offers limited free space, which forces the trajectory to include several turns in order to successfully navigate through the environment.

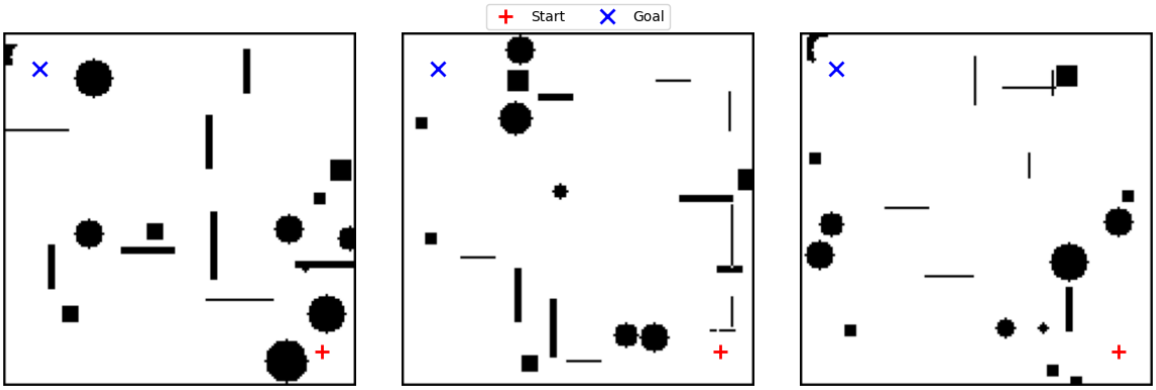

Figure 5.5: Visualization of the best individual using the cluster fitness function

5.4 Benchmarks

To evaluate whether the generated maps are suitable benchmark scenarios for controller comparison, each map was executed ten times with all three controllers (DWB, MPPI and RPP). For each configuration, the mean, minimum and maximum execution times were recorded.

Table 5.3: Gini Result Comparison

Map	1			2			3		
Controller	DWB	MPPI	RPP	DWB	MPPI	RPP	DWB	MPPI	RPP
mean	37,65	6,54	4,93	8,13	7,03	11,73	8,47	7,8	8,87
min	9,15	5,95	4,66	8	6,5	5,23	8,4	7,19	5,46
max	139,4	7,75	5,26	8,2	7,65	14,84	8,55	8,08	15,31

For the first map, the results show a clear performance drop for the DWB controller. While MPPI and RPP achieve relatively stable execution times, DWB exhibits significantly higher variance and several strong outliers. This behavior matches the intended scenario where one controller performs significantly worse than the others.

On the second map, the three controllers show noticeably different performance levels. While DWB and MPPI achieve comparable execution times, RPP requires significantly more time on average. This map therefore highlights differences in controller performance

The third map was intended to produce a scenario in which all controllers perform similarly. The results show that DWB and MPPI achieve very similar execution times. RPP,

however, exhibits slightly higher variance and occasional worse performance. Despite this deviation, the map still approximates the intended equal-performance scenario.

Overall, the maps generated using the Gini-Based fitness function successfully capture different performance patterns across the controllers. The resulting set of maps therefore provides multiple scenarios that allow meaningful comparisons between the controllers.

Table 5.4: Cluster Result Comparison

Map	1			2			3		
Controller	DWB	MPPI	RPP	DWB	MPPI	RPP	DWB	MPPI	RPP
mean	7.79	7.95	11.11	7.57	6.26	4.77	13.91	6.11	8.3
min	7.65	7.48	5.63	7.55	6,05	4.54	13.83	5.83	4.59
max	8.05	9.9	20.32	7.6	6.49	5.14	13.98	6.39	14.47

For the first map, one controller performs noticeably worse than the others. While DWB and MPPI achieve similar execution times, RPP requires considerably more time and shows higher variability.

The second map represents the intended scenario where all controllers perform differently. The results show clearly separated execution times for the three controllers, with RPP achieving the lowest average time, followed by MPPI and DWB.

On the third map, MPPI performs significantly better than the other controllers. In contrast, both DWB and RPP exhibit considerably higher execution times, resulting in a clear performance gap.

Overall, the cluster-based maps reproduce some of the intended benchmark scenarios, but not all objectives are achieved as clearly as with the Gini-based maps, indicating the impact of the fitness function on scenario quality (RQ2).

Comparing both approaches, the Gini-based fitness function appears to produce more clearly separated benchmark scenarios.

The cluster-based approach is able to generate diverse environments, but the resulting controller performance patterns are less consistent.

6 Discussion and Future Work

In the result section, we observed that the genetic algorithm is able to generate promising outcomes. However, there are also clear limitations. As shown in the graph of the minimal fitness across the 24 runs (5.1 and 5.4), the evolutionary algorithm did not fully converge, and the improvement in mean minimal fitness slowed down after only a few generations.

This is unlikely due to limitation because of the chosen encoding. Each individual consists of three 150×150 maps, resulting in 67500 values, or roughly 2^{67500} possible combinations. Rather, it appears that exploration is limited by the genetic operators. One contributing factor may be that the crossover operator is not complex enough. Since the crossover only redistributes existing maps among individuals, good individuals tend to become very similar, and the algorithm effectively explores only a small subset of the search space. This is also evident in the best individual from the Gini run, where Map 2 and Map 3 differ only lightly due to mutation.

As individuals converge, the algorithm is likely to remain in a local optimum. Beyond this point, mutation has a larger influence on exploration than crossover. Moreover, even small mutations can substantially impact fitness. For example, by blocking entire paths, indicating that mutation could be a more critical driver of exploration and might benefit from further tuning.

These findings also provide insights relevant to the research questions:

How can an evolutionary algorithm be applied to automatically generate representative test scenarios for evaluating local planners in ROS 2?:

The application of an evolutionary algorithm shows promising results. However, the current limitations of the algorithm restricts its overall effectiveness. Further work could address these limitations. Furthermore, the approach demonstrates the feasibility of creating maps with well-defined mechanics and sufficient retry mechanism, though additional strategies could further enhance robustness.

How do different fitness functions affect the ability of the EA defined in RQ1 to generate test scenarios that highlight the performance differences between local planners in ROS 2?

The impact of different fitness functions on the algorithm's ability to generate benchmark scenarios appears limited. Both fitness functions produces similar results. As described in chapter 5, the Gini-based maps showed a slightly closer match to the desired pattern, though not to an extreme degree. To determine whether a more pronounced distinction can be achieved, either the current limitations must be addressed, or the algorithm must be run until convergence.

How does the population size influence the effectiveness of the EA defined in RQ1 to generate test scenarios that highlight performance differences between local planners in ROS 2?

As discussed in previous chapters, different population sizes were evaluated using the Gini-based fitness function. Overall, all tested configurations (20,40,60) achieved comparable levels of average minimum fitness, with no statistically significant differences observed. Similar to the choice of the fitness function, the results indicate, that limited exploration may explain why changes of other components of the algorithm have little impact on the overall minimum fitness.

How can the computational cost of repeated simulations be reduced when evolutionary algorithms require frequent evaluations?:

Parallelization appears to be an effective strategy. By separating ROS and Gazebo communication and avoiding interference, multiple individuals can be evaluated simultaneously. Additional strategies, such as memory-efficient caching or selective evaluation (e.g., dropout mechanics), may further reduce computational load.

The results of this thesis demonstrate that evolutionary algorithms can effectively generate benchmark maps for evaluating controllers in ROS 2. However, several ways exist to improve the approach in future works. Algorithmic improvements could enhance exploration and diversity. For instance, the current crossover operator only redistributes existing maps, which can lead to rapid crowding and limited structural diversity within the environments. Developing more complex or structure-aware crossover and mutation operators could increase exploration and generate more varied, realistic maps. Selective evaluation strategies, such as caching previously evaluated maps or using dropout methods to temporarily skip certain controllers may further reduce computational cost without losing quality. Simulation and evaluation efficiency also presents opportunities for optimization. By separating ROS 2 and Gazebo processes and enabling parallel evaluation of multiple individuals, the computational load can be reduced. Additional optimizations could include more robust simulation setups and careful management of memory and CPU resources.

Controller specific improvements could increase informativeness of the generated maps. Tuning the parameters of the controllers rather than relying on default settings may produce maps that better differentiate controller performance. Moreover, the framework could be extended to compare global planners or a single controller under multiple parameter configurations. Evaluating more than three controllers simultaneously is also feasible with the current approach.

Scenario realism could be improved by introducing dynamic obstacles or paths simulating pedestrian like movement, resulting in maps that more closely reflect real-world conditions. Structuring genetic operators to favor realistic map patterns could further enhance the practical relevance of the benchmarks.

Finally, expanding the set of evaluation metrics beyond Distance Traveled, such as Path Smoothness, Distance to Obstacles, or multi-objective fitness would allow a more comprehensive assessment of map quality and controller performance.

Overall, the framework presented in this work provides a flexible foundation with interchangeable components, offering multiple directions for refinement and extension.

Bibliography

[1] Open Navigation LLC, "Navigation2 concepts." `https://docs.nav2.org/concepts/index.html`, 2024. Accessed: 2026-02-22.

[2] S. Macenski, F. Martín, R. White, and J. G. Clavero, "The marathon 2: A navigation system," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2718–2725, 2020.

[3] E. Brorsson, K. Ceder, Z. Zhang, S. F. Roselli, E. Erős, M. Dahl, B. Alenljung, J. Lindblom, T. Bui, E. Dean, L. Svensson, K. Bengtsson, P.-L. Götvall, and K. Åkesson, "Infrastructure-based autonomous mobile robots for internal logistics – challenges and future perspectives," 2025.

[4] G. Fragapane, R. de Koster, F. Sgarbossa, and J. O. Strandhagen, "Planning and control of autonomous mobile robots for intralogistics: Literature review and research agenda," *European Journal of Operational Research*, vol. 294, no. 2, pp. 405–426, 2021.

[5] S. Schwaiger, L. Muster, A. Scherl, P. Trivisonne, W. Wöber, and S. Thalhammer, "Towards full autonomy in mobile robot navigation and manipulation," *e+i Elektrotechnik und Informationstechnik*, vol. 141, no. 6, pp. 400–410, 2024.

[6] B. Cybulski, A. Wegierska, and G. Granosik, "Accuracy comparison of navigation local planners on ros-based mobile robot," in *2019 12th International Workshop on Robot Motion and Control (RoMoCo)*, pp. 104–111, 2019.

[7] S. Elbouhy, A. Adamanov, P. Braun, and H. Rose, "Comparative analysis of local trajectory planning algorithms in ros2," pp. 1–6, 04 2025.

[8] T. B. Sant'Anna, M. B. Argolo, and R. T. Lima, "Comparative analysis in real environment of trajectory controllers on ros2," in *2023 Latin American Robotics Symposium (LARS), 2023 Brazilian Symposium on Robotics (SBR), and 2023 Workshop on Robotics in Education (WRE)*, pp. 308–312, 2023.

[9] J. Blank and K. Deb, "Pymoo: Multi-objective optimization in python," *IEEE Access*, vol. 8, pp. 89497–89509, 2020.

[10] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, May 2022.

[11] Open Robotics, "Ros 2 documentation: Concepts." `https://docs.ros.org/en/humble/Concepts.html`, 2023. Accessed: 2026-03-08.

[12] S. Mai, N. Traichel, and S. Mostaghim, "Driving swarm: A swarm robotics framework for intelligent navigation in a self-organized world," in *2022 International Conference on Robotics and Automation (ICRA)*, pp. 01–07, 2022.

[13] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3, pp. 2149–2154 vol.3, 2004.

[14] Open Robotics, "Ros 2 article: Node lifecycle." `https://design.ros2.org/articles/node_lifecycle.html`. Accessed: 2026-03-09.

[15] S. Macenski, A. Soragna, M. Carroll, and Z. Ge, "Impact of ros 2 node composition in robotic systems," *IEEE Robotics and Automation Letters*, vol. 8, no. 7, pp. 3996–4003, 2023.

[16] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[17] S. Macenski, T. Moore, D. V. Lu, A. Merzlyakov, and M. Ferguson, "From the desks of ros maintainers: A survey of modern amp; capable mobile robotics algorithms in the robot operating system 2," *Robotics and Autonomous Systems*, vol. 168, p. 104493, Oct. 2023.

[18] E. W. Dijkstra, *A Note on Two Problems in Connexion with Graphs*, p. 287–290. New York, NY, USA: Association for Computing Machinery, 1 ed., 2022.

[19] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *Robotics Automation Magazine, IEEE*, vol. 4, pp. 23 – 33, 04 1997.

[20] ROS Navigation Stack Contributors, "dwa_local_planner." `https://wiki.ros.org/dwa_local_planner`, n.d. Accessed: 2026-03-14.

[21] G. Williams, P. Drews, B. Goldfain, J. M. Rehg, and E. A. Theodorou, "Aggressive driving with model predictive path integral control," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1433–1440, 2016.

[22] R. C. Coulter, "Implementation of the pure pursuit path tracking algorithm," Tech. Rep. CMU-RI-TR-92-01, Carnegie Mellon University, Robotics Institute, 1992.

[23] S. Macenski, S. Singh, F. Martin, and J. Gines, "Regulated pure pursuit for robot path tracking," 2023.

[24] R. Kruse, S. Mostaghim, C. Borgelt, C. Braune, and M. Steinbrecher, *Computational Intelligence: A Methodological Introduction.* Texts in Computer Science, Cham: Springer International Publishing, 2022.

[25] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[26] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, G. N. Yannakakis, and C. Grappiolo, "Controllable procedural map generation via multiobjective evolution," *Genetic Programming and Evolvable Machines*, vol. 14, pp. 245–277, June 2013.

[27] A. Pech, M. Masek, C.-P. Lam, and P. Hingston, "Game level layout generation using evolved cellular automata," *Connection Science*, vol. 28, no. 1, pp. 63–82, 2016.

[28] A. Baldwin, S. Dahlskog, J. M. Font, and J. Holmberg, "Mixed-initiative procedural generation of dungeons using game design patterns," in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 25–32, 2017.

[29] N. Promkaew, S. Thammawiset, P. Srisan, P. Sanitchon, T. Tummawai, and S. Sukpancharoen, "Development of metaheuristic algorithms for efficient path planning of autonomous mobile robots in indoor environments," *Results in Engineering*, vol. 22, p. 102280, 2024.

[30] Y. Xue, "Mobile robot path planning with a non-dominated sorting genetic algorithm," *Applied Sciences*, vol. 8, no. 11, 2018.

[31] K. Kim, J. Lee, and J. Jeon, "Performance analysis of dwb, mppi, and regulated pure pursuit, rotate shim for ros2-based amr navigation on simulation," pp. 1–5, 10 2025.

[32] H. Yuan, H. Li, Y. Zhang, S. Du, L. Yu, and X. Wang, "Comparison and improvement of local planners on ros for narrow passages," in *2022 International Conference on High Performance Big Data and Intelligent Systems (HDIS)*, pp. 125–130, 2022.

[33] J. Wen, X. Zhang, Q. Bi, Z. Pan, Y. Feng, J. Yuan, and Y. Fang, "Mrpb 1.0: A unified benchmark for the evaluation of mobile robot local planning approaches," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 8238–8244, 2021.

[34] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[35] M. T. Catalano, T. L. Leise, and T. J. Pfaff, "Measuring resource inequality: The gini coefficient," *Numeracy*, vol. 2, no. 2, p. Article 4, 2009.

[36] T. O. Kvålseth, "Rank-based inequality measures: an alternative to gini's index," *International Review of Economics*, vol. 72, p. 2, Nov 27 2024.

[37] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American Statistical Association*, vol. 47, no. 260, pp. 583–621, 1952.

Declaration of Authorship

Thesis:

Name: Surname:

Date of birth: Matriculation no.:

I herewith assure that I have written the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or according to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Furthermore, I confirm that I am aware that the use of content (including but not limited to text, figures, images and code) generated by artificial intelligence (AI) in the thesis must be disclosed. In those cases I have specified the AI system used, I have marked the specific sections of the thesis where AI-generated content was used, and I have provided a brief explanation of the level of detail at which the AI system was used to generate the content. I also stated the reason for using the tools.

I assure that even if a generative AI system has been used, the scientific contribution has been made entirely by myself.

Lasse Slaar Magdeburg, March 21, 2026