Final Project - Algorithmic Robotics and Motion Planning

Yehonathan Cohen

May 26, 2024

1 Abstract

This project addresses a variant of the Coverage Path Planning (CPP) problem. It aims to tackle this issue using a genetic algorithm-based approach, experimenting with a range of techniques for implementing a genetic algorithm to solve the problem, and presenting comparisons between these techniques. Some of these techniques with a slight modification and adjustments can be applied to real world multi-agents problems such the cleaner robots problem and the bus routing problem.

2 Problem Definition

The problem is a variation of the Coverage Path Planning problem.

2.1 The settings

The setting is a 2D workspace with k disc-shaped robots moving in a bounded environment containing polygonal obstacles. Each robot has a start point and an end point. A *valid path* for a robot is one that connects its start point to its end point without colliding with any obstacles. This path is composed of points, referred to as *visit points*, connected by straight lines. To define the problem, we use a grid overlay on the environment that divides it into uniformly sized square *cells.* The *cell size* is the length of an edge of each cell.

2.2 No robot collision assumption

We assume that robots can collide without invalidating their paths, which simplifies the algorithm and reduces running time. This is crucial for genetic algorithms, which require extensive hyperparameter tuning and many experiments. Though simplistic, this assumption is valid for many real-world problems.

For example, consider k drones flying low between buildings to scan areas. They can collide because they operate at different altitudes. Each drone follows a path with designated visit points for scanning. Our goal is to ensure every cell is scanned by at least one drone and to minimize redundant scanning, so no two drones scan the same cell.

2.3 The goal

Building on the motivation above, the objective is to determine a *valid path* for each robot, maximizing the number of cells that have visit points unique to a single robot.

This can be formalized as: given k robots, P the set of all valid solution of the form $(p_1, ..., p_k)$ where p_j is a *valid path* for the j'th robot, which is a sequence of *visit points*, and C the set of all *cells*, find:

$$\underset{(p_1,...,p_k)\in P}{argmax} \; |\{c \in C \mid |\{j \in [k] \mid \exists v \in p_j \mid v \in c\}| = 1\}|$$

3 The Method

3.1 Overview

A genetic algorithm consists of several components. First, an *initial population* is created, which is a set of individuals, each representing a valid solution to the problem—in our case, a set of *valid paths*, one for each robot. Then, for a specified number of evolution steps, an evolution iteration is performed to create the next generation, replacing the current population. Each evolution step includes the following stages:

1. *Fitness function*: Compute a fitness function for each individual, representing the value of the objective function.
2. *Elite selection*: Select the individuals with the highest fitness values and copy them to the next generation.
3. *Crossover*: Generate additional individuals for the next generation by randomly selecting two parent individuals from the previous generation and combining them. Individuals with higher fitness values have a greater chance of being selected as parents.
4. *Mutation*: Randomly select some of the new individuals and apply random changes to them.

Finally, return the solution with the highest fitness value.

In this paper, we explore various options for each component of the genetic algorithm. Besides the different high-level choices for each step, there are also additional hyperparameters. The overall steps and hyperparameters are illustrated in Figure 1. In the following sections, we will describe the implementations tested for each component, followed by a presentation of the results.

3.2 Initial Population

Two strategies where explored for initializing the first population.

In both cases, the first step is to create a roadmap for each robot, similar to the PRM algorithm. This involves randomly sampling n collision-free points for each robot and connecting each point to its m nearest neighbors if the connecting edge is collision-free.

We refer to one approach as *random point initialization*, where for each robot we randomly sample a point uniformly from its roadmap. Subsequently, if a path exists, it is formed by concatenating the shortest path from the start point to the random point with the shortest path from the random point to the end point.

The second approach, termed *shortest path initialization*, directly connects the start point to the end point via the shortest path. Consequently, in this scenario, all individuals in the initial population are identical.

While the first approach offers a wider range of solutions and a stronger starting point, it may also result in undesirable paths that are challenging to rectify later.

A hyperparameter that is related to this step is the size of the population.

3.3 Fitness Value

For the fitness value, two approaches were explored.

The first approach, dubbed *cells and lengths*, combines two objectives: (1) maximizing the number of cells that have a robot with a *visit point* in them, and (2) minimizing the total length of the paths. Despite its intuitive nature, balancing these two objectives proved challenging, as there is obviously a trade-off between them. As a result, this approach was ultimately discarded.

The second approach, previously discussed in relation to our goal definition defines the fitness value as the number of cells that contain visit points from a unique robot. In line with the notation previously introduced, for a given individual $(p_1, ..., p_k)$, the fitness value is:

$$|\{c \in C \mid |\{j \in [k] \mid \exists v \in p_j \mid v \in c\}| = 1\}|$$

Additionally, an important aspect of this approach involves starting with a larger cell size and gradually reducing it (by factor 2) throughout the iterations until it reaches the actual cell size used to compute the final fitness values. The rationale behind this technique is to encourage the robots to explore diverse areas. By initially considering larger cells, the robots may explore distinct zones and diverge in various directions from one another. As the iterations progress, the cell size is decreased, with the expectation that the robots will gradually fill the different zones they have identified. This technique necessitates setting three hyperparameters:

1. *Cell size decrease interval*: The number of steps without improvement in the maximum fitness value before we decrease the cell size.
2. *Min cell size*: The final size to which the cell size is reduced.
3. *Final steps number*: To ensure that fitness values in the last iterations are computed according to the *minimum cell size*, the cell size is set to the *minimum cell size* in the final steps if it hasn't already reached this value.

3.4 Elite Selection

In this stage, the genetic algorithm selects individuals with the highest fitness values and duplicates them for the next generation. The only hyperparameter to be determined in this step, the *elite portion*, is the portion of individuals selected for this process.

3.5 Crossover

For the crossover operation, again two methods were considered.

In both cases, we select the parents of each new individual using the fitness proportionate selection[1]

The first crossover option, which we refer to as *Merge*, entails combining paths for the same robot. To elaborate, considering two parents denoted as $parent_0$ and $parent_1$, for each robot path to be created, we generate a path that commences with the beginning of the $parent_0$ robot's path up to a randomly selected point in this path. Subsequently, we proceed with the shortest path to another randomly selected point in the $parent_1$ robot's path, concluding with the continuation of the $parent_1$ robot's path to the endpoint.

The second crossover option, which we refer to as *Copy*, involves randomly selecting a parent between $parent_0$ and $parent_1$ for each robot. Subsequently, the entire robot path is copied from this selected

[1] https://en.wikipedia.org/wiki/Fitness_proportionate_selection

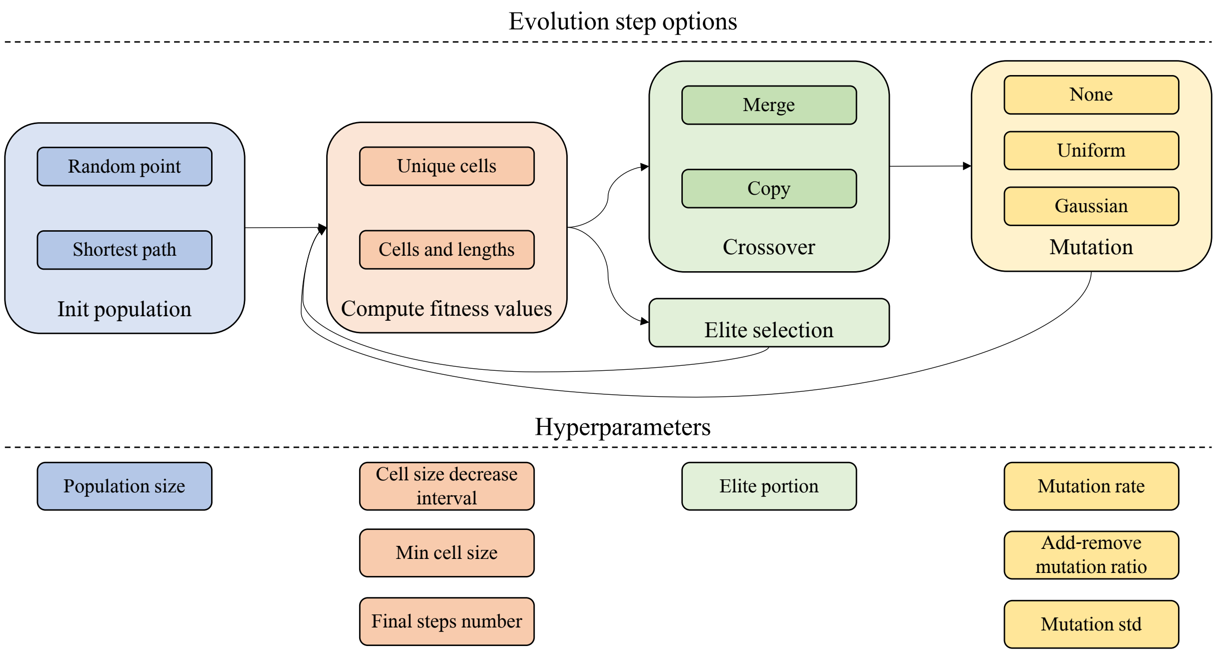

Figure 1: The different options for the genetic algorithm and the corresponding hyperparameters.

parent. It's important to note that, in this option, an individual comprises multiple robot paths, each taken from either $parent_0$ or $parent_1$.

3.6 Mutation

For each individual from the crossovers, there are three options for mutations:

1. *None*: In this option, no mutation is applied. The portion of individuals without mutation operator is determined by the hyperparameter *mutation rate*.

2. *Uniform*: In this approach, we modify a robot's path by randomly sampling a point from the robot's roadmap. Subsequently, we sample two points from the robot's existing path and connect the first point to the randomly sampled point, followed by connecting the randomly sampled point to the second point using the shortest paths available.

3. *Gaussian*: In this method, we start by selecting a random point p_1 from the robot's path. Subsequently, we sample a new point p_2 from a Gaussian distribution centered at p_1 and with a standard deviation which is determined by another hyperparameter *mutation std*. Next, we substitute p_1 with a point p_3 from the robot's roadmap, chosen as the closest point to p_2. To prevent an excessive increase in the number of points, we employ a technique for removing points. Here, we randomly select two points from the robot's path and connect them via the shortest paths, effectively shortening the path. Consequently, another hyperparameter is defined: the *add remove mutation ratio*, representing the probability of adding a new point versus shortening the path.

4 Experiment

4.1 Baseline

4.1.1 Framework and Code

The algorithm was implemented using the Discopygal framework[2]. The code, along with the results, is available on GitHub[3]. The primary object in this code is the *CoveragePathPlanner* class, which extends Discopygal's *Solver*. This class includes a method named *load_scene* that implements the genetic algorithm process depicted in Figure 1. Additionally, there is a *main.py* file for running experiments, detailed in the following sections. Instructions for running the script are available in the repository's README file.

4.1.2 Scenes

For the experiments, we used three different scenes with varying numbers of disc-robots, start and end locations, and polygonal obstacles. The scenes are shown in Figure 2 and can be found on GitHub.

In Scene 1, there are two robots in a simple environment with a single obstacle. Each robot's start point is the other robot's end point.

[2] https://www.cs.tau.ac.il/~cgl/discopygal/docs/index.html
[3] https://github.com/yonicohen96/cleaner-robot-ga

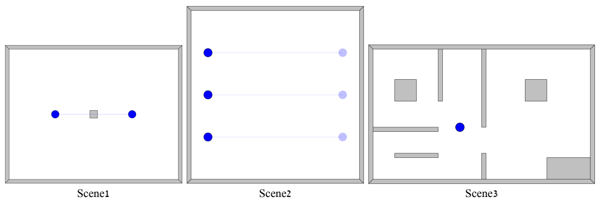

Figure 2: The scenes used in our experiments.

In Scene 2, there are three robots in an even simpler workspace without obstacles, each with different start and end points.

Scene 3 is more complex, with obstacles and four robots, all sharing the same start and end point.

4.1.3 Experiment hyperparameters

In each scene, we established the roadmap for individual robots following the procedure outlined in 3.2 , using n = 1000 samples and m = 15 neighbors. We fixed a set of hyperparameters, and across every experiment, we investigated the outcomes of various alterations to a particular hyperparameter. The base hyperparameters are: *population size* = 10, *min cell size* = 2.0, *cell size decrease interval* = 5, *final steps num* = 10, *random point initialization* = 0, *elite proportion* = 0.2, *crossover merge* = 0, *mutation rate* = 0.5, *mutate gauss* = 1, *add remove mutation ratio* = 0.8, *mutation std* = 2,

Each experiment was repeated three times, with assessments conducted to gauge the mean and standard deviation of the best fitness values over evolutionary steps, as well as the average time elapsed.

4.2 Results and Observations

4.2.1 population size

As illustrated in Figure 3, a larger population size generally leads to higher fitness values. This is because a greater number of individuals provides more opportunities for combinations and mutations, increasing the likelihood of individuals with higher fitness values. However, this also significantly increases the running time, as depicted in Figure 4. Additionally, for simpler scenarios such as Scene1, a larger population size does not offer any advantages.

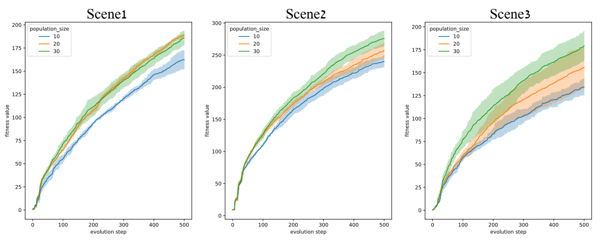

Figure 3: Results for population_size experiments.

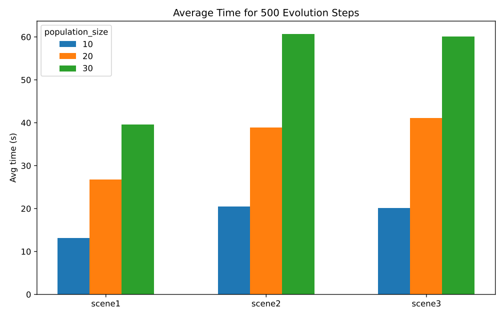

Figure 4: Running times for population_size experiments.

4.2.2 cell size decrease interval

The results for the hyperparameter *cell size decrease interval*, as described in 3.3, show that for simpler scenarios like Scene1, lower values are preferable. However, for more complex scenarios such as Scene3, lower values of this hyperparameter initially yield higher fitness values, but after several steps, higher values achieve better fitness outcomes. This supports the hypothesis that initially using larger cells allows the robots to explore distinct zones and diverge in various directions from one another as demonstrated in 6. The average running times for all values of this hyperparameter are similiar.

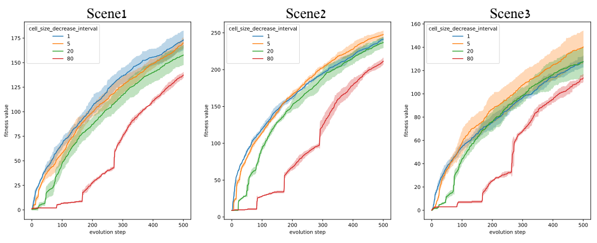

Figure 5: Results for cell_size_decrease_interval experiments.

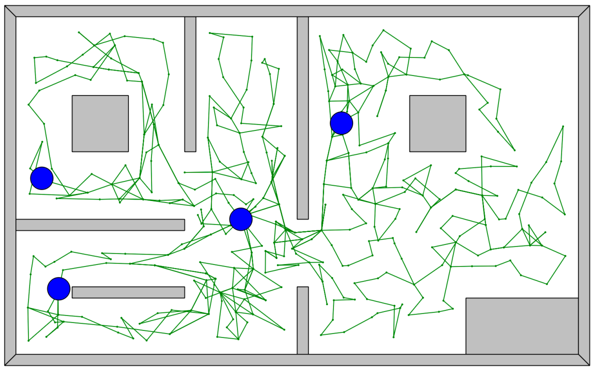
Figure 6: Example result for using cell size decrease interval paths of 5.

4.2.3 random point initialization

Settings this binary hyperparameter to 0 means using the random point strategy for the first population initialization, and 1 for the shortest path initialization, both described in 3.2. As illustrated by the graphs in figure 7, random initialization benefits from a better starting point. This advantage is not significant in the initial steps only because the cell size is large at the beginning.

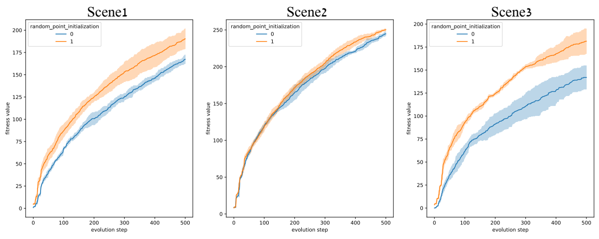

Figure 7: Results for random_point_initialization experiments.

4.2.4 elite proportion

For the elite proportion hyperparamter, the experiments shown in figure 8 indicate that for Scenes 1 and 3, selecting values between 0.2 and 0.5 yields no significant difference. This is likely because the top two individuals in these scenes consistently produce the best offspring during crossover operations. In Scene 2, however, there is a slight difference. This might be attributed to the greater freedom of path choices in this scenario, where a lower elite proportion allows for more crossovers and mutations, thereby exploring a wider variety of paths and combinations. It is important to note that a lower elite proportion results in a higher number of mutations and crossovers, which in turn increases the running time.

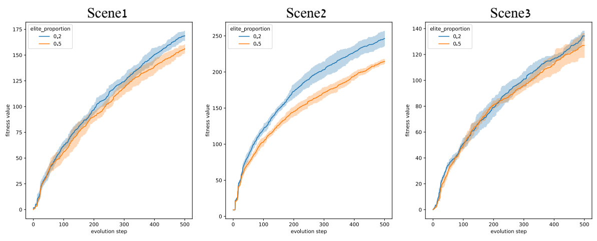

Figure 8: Results for elite_proportion experiments.

4.2.5 crossover merge

The values for this hyperparameter are 0 and 1, corresponding to choosing between *merge* and *copy* for the crossover operator, as described in 3.5. This hyperparameter yields interesting results shown in figure 9, as different scenes produce different outcomes. For Scenes 1 and 3, it is more effective to use the copy strategy, whereas for Scene 2, the merge strategy is preferable. This can be explained by the fact that Scene 2 offers more path options, benefiting from exploration. The merge strategy creates new paths by combining two existing paths, thus exploring more possibilities compared to simply copying existing ones.

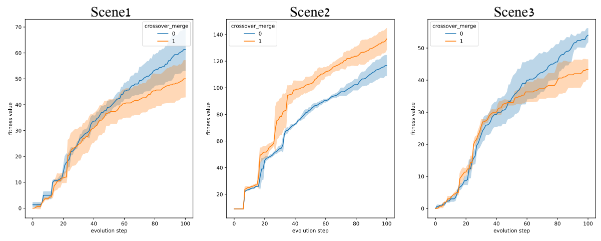

Figure 9: Results for crossover_merge experiments.

4.2.6 mutation parameters

The mutation-related hyperparameters were described in 3.6. The experimental results for these hyperparameters (figures 10, 11, 13, 14) demonstrate that for the mutation definitions and values tested, more radical mutations lead to higher maximum fitness values. This is because more radical mutations enable faster exploration of new regions. However more radical mutations leads to a significant increase in the running time.

For the *mutation rate* hyperparameter, increasing the number of mutations up to a certain threshold enhances exploration. However, beyond this threshold, a high mutation rate can compromise the retention of good paths from the previous generation, resulting in overly random paths, as observed in the later stages of the Scene 3 experiment.

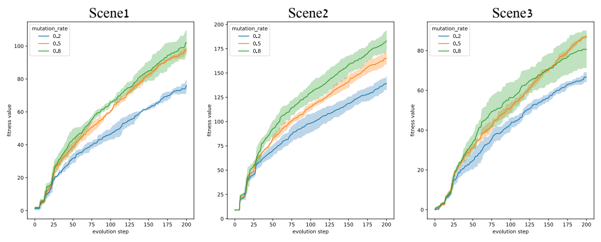

Figure 10: Results for mutation_rate experiments.

The *mutate gauss std* parameter selects the mutation operator from those described in 3.6, with 0 representing the *uniform* method and 1 representing the *gaussian* method. It's evident that sampling uniformly across the environment facilitates faster exploration of new areas compared to sampling from a Gaussian distribution centered around an already selected point. However the running time for the uniform method is much larger.

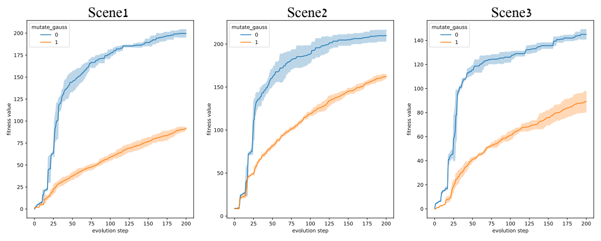

Figure 11: Results for mutate_gauss experiments.

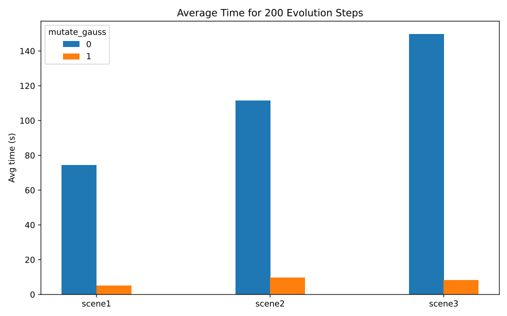

Figure 12: Running times for mutate_gauss experiments.

The findings from the experiments concerning the *add remove mutation ratio* hyperparameter indicate that, for the values tested, the regularization in shortening paths using the reduction operator only only leads to lower fitness values.

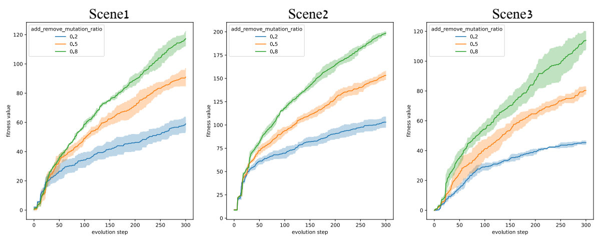

Figure 13: Results for add_remove_mutation_ratio experiments.

Regarding the *mutation std* hyperparameter, we observe higher fitness values with higher parameter values. This phenomenon, as discussed earlier, can be attributed to greater exploration facilitated by larger parameter values.

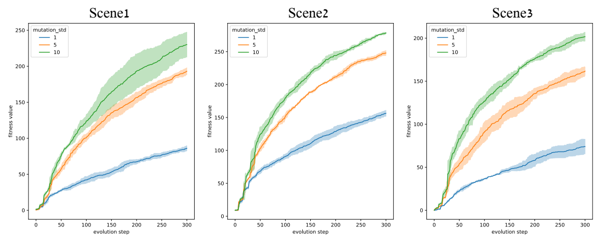

Figure 14: Results for mutation_std experiments.

5 Future Work

We've introduced a range of genetic algorithm implementations for addressing the coverage path planning problem. We've showcased the diversity of options available for each operator and provided insights into the outcomes associated with different choices of hyperparameters. Our results reveal that in scenarios where there's greater flexibility in path selection, opting for implementations with higher exploration rates enhances fitness values, albeit at the expense of increased running time.

Due to the inherent flexibility of genetic algorithms, there are numerous approaches we can take to improve the results. One simple example is modifying the mutation operators to sample from cells that haven't been visited yet.

Additionally, we could adjust the fitness function. For example, we could include a term that incentivizes paths leading to a similar workload distribution among the robots, ensuring that the path lengths are more balanced.