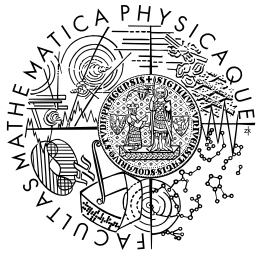


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Peter Lakatoš

Collision Avoidance in Computer Games

Department of Software Engineering

Supervisor of the master thesis: Mgr. Jakub Gemrot, Ph.D.

Study programme: Computer Science

Prague 2024

I declare that I carried out this master thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

First, I would like to thank my supervisor, Mgr. Jakub Gemrot, Ph.D., for always finding time for a discussion, detailed feedback, and continuous encouragement that motivated me during working on this thesis. My appreciation also goes to my second supervisor, Juraj Blaho, for his valuable insights on this topic, without which this thesis would not exist.

My deepest gratitude goes to my parents and the rest of my family for their endless support, without which I would not be able to go through my studies.

Title: Collision Avoidance in Computer Games

Author: Bc. Peter Lakatoš

Department: Department of Software Engineering

Supervisor: Mgr. Jakub Gemrot, Ph.D., The Department of Software and Computer Science Education (KSVI)

Abstract: Collision avoidance for autonomous agents has been a widely researched topic for the past couple of decades. Modern solutions act as purely reactive techniques that create various problems, such as agents being stuck in various scenarios. The aim of this thesis was to explore a new way of solving collision avoidance for humanoid agents using genetic algorithms to search local space multiple steps ahead of the current simulation state.
The application is capable of running multiple predefined test scenarios and logging the results of each run. The application provides two possible ways of seeing the results, either visually observing the scenario run or plotting the results logged by the application.
The overall design of the application is general enough to allow simple modification to existing scenarios or creation of new ones. It is also possible to modify an existing genetic algorithm with new operators with minimal effort.
The results show that even though various configurations of the implemented genetic algorithm perform similarly, there are some outstanding winners that might bring an alternative possibility to the already existing collision avoidance methods.

Keywords: Collision avoidance, Genetic algorithms, Agents

Název práce: Vyhýbání se kolizím při pohybu agentů v prostředí počítačových her

Autor: Bc. Peter Lakatoš

Katedra: Softwarové a datové inženýrství

Vedoucí bakalářské práce: Mgr. Jakub Gemrot, Ph.D., Katedra softwaru a výuky informatiky (KSVI)

Abstrakt: Vyhýbanie sa kolíziám autonómnych agentov je za posledných niekoľko desaťročí predmetom rozsiahleho výskumu. Aktuálne riešenia fungujú ako čisto reaktívne techniky, ktoré majú za následok niekoľko problémov, ako napríklad uviaznutie agentov v rôznych situáciách. Cieľom tejto práce bolo preskúmať novú metódu riešenia vyhýbania sa kolíziám pre humanoidných agentov za pomoci genetických algoritmov, ako spôsob prehľadávania lokálneho priestoru niekoľko simulačných krokov dopredu.
Aplikácia je schopná spustiť viacero preddefinovaných testovacích scenárov a zaznamenať výsledky každého behu. Aplikácia takisto umožňuje sledovať výsledky dvoma rozličnými spôsobmi, buď vizuálne pozorovať beh testovacieho scenára, alebo vykresliť zaznamenané výsledky v podobe grafov.
Celkový dizajn aplikácie je dostatočne obecný na to, aby umožňoval jednoduché modifikácie existujúcich testovacích scenárov, alebo vytvorenie úplne nových. Takisto je možné modifikovať existujúci genetický algoritmus pomocou nových operátorov s minimálnym úsilím.
Výsledky ukazujú, že aj keď rôzne konfigurácie implementovaného genetického algoritmu fungujú podobne, existujú niektoré konfigurácie vyčnievajúce z davu, ktoré by mohli priniesť alternatívnu možnosť k už exitujúcim metódam na vyhýbanie sa kolíziám.

Klíčová slova: Vyhýbanie sa kolíziám, Genetické algoritmy, Agenti

Contents

1 Introduction

Collision avoidance has been the broad area of study for the past several decades. We can see its application in numerous fields, such as robotics, or in a virtually simulated environments like computer games.

In virtually simulated environments, we often have to deal with very diverse conditions and circumstances. In some situations, we might need to simulate hundreds of agents, while in others there are only several of them. We also need to take into account the possibility of external factors, such as players. Proper collision avoidance algorithm must take into account all these elements, as it provides not only correct functionality but also more realistic immersion.

Currently, there is no bulletproof solution that is error-free in every possible scenario. Many modern state-of-the-art methods act as purely reactive techniques, which might result in unnatural behaviour of the agents and break the overall immersion for the player.

The aim of this thesis is to explore some less commonly used techniques for solving collision avoidance for humanoid agents using genetic algorithms.

1.1 Background and Motivation

This thesis will focus on virtually simulated environments, such as computer games with humanoid agents. This means that agents will have human-like limitations on their physical traits such as size, speed, and acceleration.

Collision avoidance is part of the agent navigation process. For the navigation process to occur, each agent needs to have a destination. After that, agents first plan their paths (which usually results in the shortest path) and then navigate along that path. The collision avoidance algorithm should react to static as well as dynamic obstacles (such as other agents) during navigation. Navigation eventually means setting the velocity for the agent at each simulation step. This velocity is often the result of the collision avoidance algorithm. Each agent has its preferred velocity, which usually follows directly the pre-planned path. The resulting velocity during navigation might deviate from the agent's preferred velocity because of the need to avoid obstacles. The search for this velocity can also be viewed as a multi-objective optimisation problem, as it needs to satisfy multiple criteria. For example, it needs to be as close as possible to the agent's preferred velocity and should also avoid as many obstacles as possible. The other way to look at this problem is to search the local space area to find the most sufficient velocity among all other velocities that will move the agent closer to the destination. To find a solution, we can use genetic algorithms, as they have proved to be successful in

this domain of problems.

1.2 Problem statement

Modern state-of-the-art collision avoidance algorithms use purely reactive techniques, which often results in unrealistic behaviour of agents. Some examples of such behaviour might be that agents always prefer the shortest path possible, which might result in overcrowding in narrow corridors, moving too close to the walls (often referred to as wall-hugging), and getting stuck.

This thesis aims to explore possible solution to these problems, using genetic algorithms as a local space search algorithm that will explore viable paths that an agent can take to successfully navigate to the destination. The metrics that will be used to evaluate the quality of possible paths are the following:

1. How far from destination would the agent be at the end of a given path.
2. How many steps would it take for the agent to reach the destination, if moving along a given path.
3. Number of collisions on a given path.
4. How smooth a given path is in terms of jerk-cost.

We have the following requirements for this thesis:

(R1) Implement collision avoidance algorithm using genetic algorithm.

(R2) Provide multiple scenarios and perform experiments to test the implemented collision avoidance algorithm.

(R3) Run experiments on multiple configurations of the implemented genetic algorithm.

(R4) The metrics defined for the evaluation of the path are captured and saved from the experiments.

(R5) The experiments are visually observable in simulation.

(R6) The results of the experiments are captured, analysed, and discussed in the form of graphs.

2 Collision avoidance

In this chapter, we first go through the modern state-of-the-art collision avoidance methods, such as Velocity Obstacles (VO), Reciprocal velocity Obstacles (RVO), and Optimal reciprocal collision avoidance (ORCA) in Section 2.1. Then we look at the common problems associated with these methods in Section 2.2

2.1 Modern state-of-the-art methods

In this section we first explain the collision avoidance algorithm called *Velocity obstacles* (also referred to as VO) introduced by Fiorini and Shiller (16). Then, we show the iteration of this algorithm called *Reciprocal velocity obstacles* (also referred to as RVO), which is among the most widely used collision avoidance algorithms. The idea behind both of these algorithms is that agents are independent and decentralised, meaning that they are making their own decisions based purely on observation of other agents. At the end of this section we explain another iteration of VO called Optimal reciprocal collision avoidance (also referred to as ORCA).

2.1.1 Velocity obstacles

In this method, we assume an environment consisting of an agent A and a moving obstacle B, both of circular shape. We also assume that for both the agent and the obstacle, we know their corresponding positions and velocities. Let v_a represent the velocity of the agent A, and v_b represent the velocity of the obstacle B. As stated in the paper by Fiorini and Shiller (16), to compute VO, we first need to map B into the *Configuration Space* of A, by reducing A to the point $\hat{A}$ and expanding B by the radius of A to $\hat{B}$. We then define the *Collision Cone* $CC_{A,B}$ (Figure 2.1) consisting of the relative velocities between $\hat{A}$ and $\hat{B}$ that collide with each other. It can be formulated as follows:

$$CC_{A,B} = \{v_{A,B} \mid \lambda_{A,B} \cap \hat{B} \neq \emptyset\}$$

where $v_{A,B}$ is the relative velocity of $\hat{A}$ with respect to $\hat{B}$ and $\lambda_{A,B}$ is the line of $v_{A,B}$.

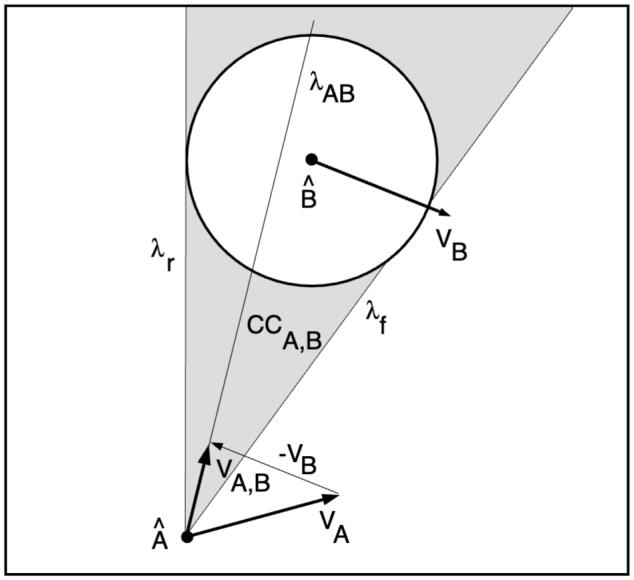


Figure 2.1 Collision cone $CC_{A,B}$. From *Motion Planning in Dynamic Environments Using Velocity Obstacles*, by Fiorini and Shiller (16)

To modify this into absolute velocities, we can translate $CC_{A,B}$ by v_B, which is done by the Minkowsi sum. This defines the Velocity Obstacle set VO (Figure 2.2) as following:

$$VO = CC_{A,B} \oplus v_B$$

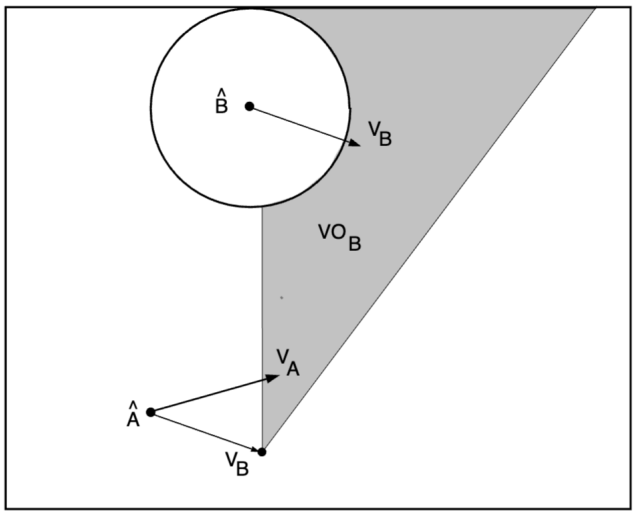


Figure 2.2 Velocity Obstacle set VO. From *Motion Planning in Dynamic Environments Using Velocity Obstacles*, by Fiorini and Shiller (16)

If the agent selects a velocity outside the set VO, it will avoid collision with the obstacle. To generalise this method for multiple obstacles, $B_1...B_n$, we calculate the union of the corresponding velocity obstacles sets (Figure 2.3) as follows:

$$VO = \bigcup_{i=1}^{n} VO_{B_i}$$

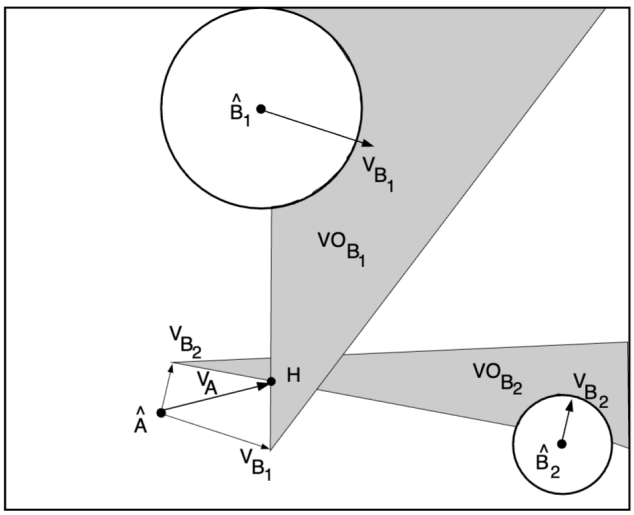


Figure 2.3 Velocity Obstacle set VO for obstacles B1 and B2. From *Motion Planning in Dynamic Environments Using Velocity Obstacles*, by Fiorini and Shiller (16)

The velocity obstacle algorithm is not perfect and has some drawbacks (which will be more closely described in Section 2.2). The most serious one is that it may produce trajectories that are not smooth and result in oscillating agents. Therefore, Berg, Lin, and Manocha (17) presented the new iteration called *Reciprocal velocity obstacles*, which will be described in Section 2.1.2.

2.1.2 Reciprocal velocity obstacles

The idea behind the reciprocal velocity obstacles method is simple. Instead of agents selecting their velocities out of their corresponding VO sets, they now share responsibility. We explain this principle analogously as presented in the paper *A survey on Velocity Obstacle paradigm* presented by Vesentini, Muradore and Forini (18).

Consider two agents A and B facing each other and moving in opposite directions. The agent A moves with velocity v_A such that $v_A \in VO_{A,B}$ and the agent B moves with velocity v_B such that $v_B \in VO_{B,A}$. Both agents decide to use *Velocity obstacle*

algorithm to avoid each other. The agent A selects a new velocity $v_{A-new} = v_A + w$, where $w \in \mathbb{R}^2$ such that $v_{A-new} \notin VO_{A,B}$. The agent B achieves the same by selecting $v_{B-new} = v_B + w, v_{B-new} \notin VO_{B,A}$. Both of the new selected velocities are not optimal in terms of how distant they are from the agents' preferred velocities. To reach the optimal solution, agents now share the responsibility of avoiding each other, meaning that each agent will go halfway to the side only as much as necessary to ensure that they avoid each other. For agent A, this means that instead of calculating $v_{A-new} = v_A + w$, it selects $v_{A-new} = v_A + \frac{w}{2}$. The same applies to agent B. We define the reciprocal velocity obstacle set $RVO_{A,B}$ for the agent A as following:

$$RVO_{A,B} = \{v_{A-new} \mid 2v_{A-new} - v_{A-new} \in VO_{A,B}\}$$

Definition for the agent B is analogous. For agents to successfully avoid each other, they both must select a velocity outside of their RVO sets. Geometrically, the $RVO_{A,B}$ cone can be represented as the $VO_{A,B}$ cone translated by $\frac{v_A+v_B}{2}$, which is shown in Figure 2.4.

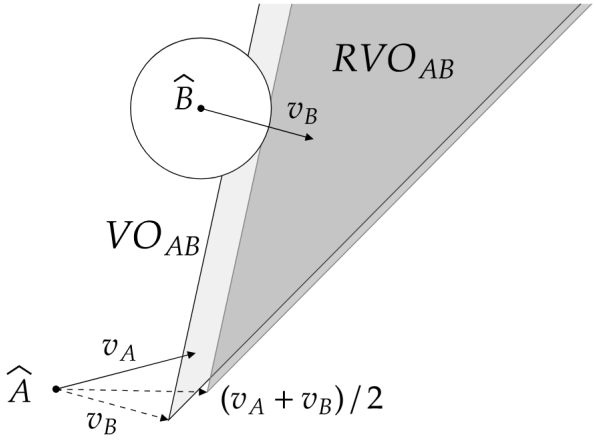


Figure 2.4 The geometrical representation of $RVO_{A,B}$. From *A survey on Velocity Obstacle paradigm*, by Vesentini, Muradore and Forini (18)

To generalise this method among multiple agents we use approach presented by Vesentini, Muradore and Forini (18), by considering the agents $A_1 ... A_n$ such that each agent has its preferred velocity v_{A_i-pref} and goal destination d_{A_i}. For the agent A_i to avoid collision, we select a new velocity such that

$$v_{A_i-new} = \underset{v \notin RVO_{A_i}}{\operatorname{argmin}} \parallel v - v_{A_i-pref} \parallel_2$$

where RVO_{A_i} is given by

$$RVO_{A_i} = \bigcup_{j \neq i}^{n} RVO_{A_i, A_j}$$

2.1.3 Optimal reciprocal collision avoidance

Optimal reciprocal collision avoidance (referred to as *ORCA*) presented by Berg, Guy, Lin and Manocha (19) is another iteration of collision avoidance based on the *Velocity obstacles*.

Consider two agents A and B with their current velocities v_A and v_B respectively. As described in the paper *A survey on Velocity Obstacle paradigm* presented by Vesentini, Muradore and Forini (18), based on the *Velocity obstacles* method from Section 2.1.1, we can construct the $VO^{\tau}_{A,B}$ cone that denotes all velocities of the agent A that cause a collision with the agent B within a time horizon $\tau > 0$. Formally, it is defined as the following:

$$VO^{\tau}_{A,B} \triangleq \{v \mid \exists t \in [0, \tau], tv \in D(x_B - x_A, r_A + r_B)\}$$

where x_A, r_A and x_B, r_B are positions and radii of the agents A and B respectively, and D is a disc with a center in $x_B - x_A$ with radius equal to $r_A + r_B$. This is shown in Figure 2.5.

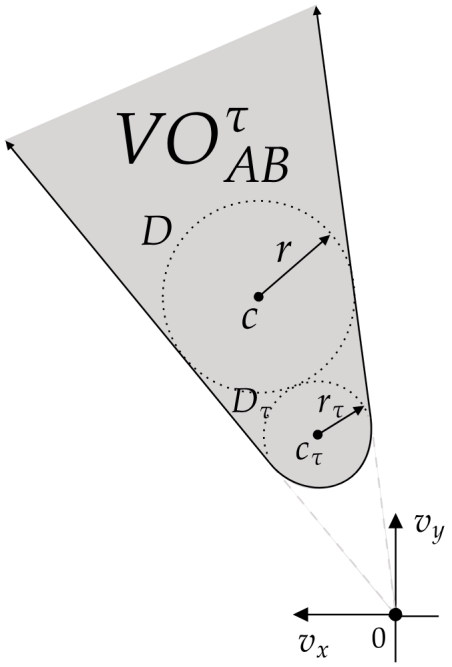


Figure 2.5 The geometrical representation of $VO^{\tau}_{A,B}$, $r = r_A + r_B$ with the center in $c = x_B - x_A$. From *A survey on Velocity Obstacle paradigm*, by Vesentini, Muradore and Forini (18)

If agents A and B collide before time τ, then $v_{A,B} = v_A - v_B \in VO^{\tau}_{A,B}$. We can define the vector w that originates in $v_{A,B}$ and ends at the closest point of the cone $\delta VO^{\tau}_{A,B}$ as follows:

$$w = (\underset{v \in \delta VO^{\tau}_{A,B}}{\operatorname{argmin}} \parallel v - v_{A,B} \parallel_2) - v_{A,B}$$

Lastly, we denote n as the normal vector facing in direction w at point $v_{A,B} + w \in \delta VO^{\tau}_{A,B}$. We can define a set of collision-free velocities for the agent A as the half-plane with the origin at $v_A + \frac{w}{2}$ and the direction of w. Formally, this is written as following:

$$ORCA^{\tau}_{A,B} \triangleq \{v \mid (v - (v_A + \frac{w}{2})) * n \geq 0\}$$

Definition for the agent B is analogous. The geometric representation can be seen in Figure 2.6

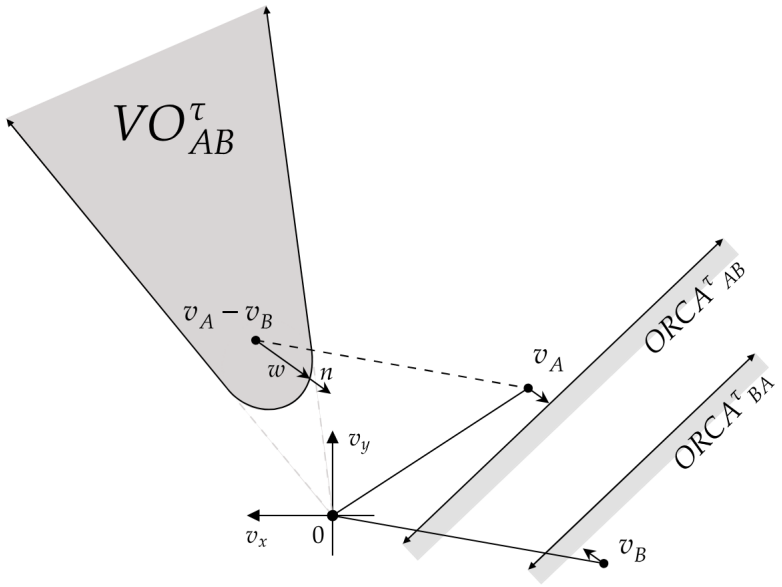


Figure 2.6 The geometrical representation of $ORCA^{\tau}_{A,B}$ and $ORCA^{\tau}_{B,A}$. From *A survey on Velocity Obstacle paradigm*, by Vesentini, Muradore and Forini (18)

To generalise this method among multiple agents we use approach presented by Vesentini, Muradore and Forini (18), by considering the agents $B_1...B_m$, then the set of collision-free velocities for the agent A is defined as

$$ORCA^{\tau}_{A,B} \triangleq D(0, v_A^{max}) \cap \bigcap_{i=1}^{m} ORCA^{\tau}_{A_i,B_i}$$

where $D(0, v_A^{max})$ is disc with centre in the origin and radius size of the maximum reachable agent speed. Then, the agent's A collision-free velocity is defined as

$$v_A^{new} = \underset{v \in ORCA^{\tau}_A}{\operatorname{argmin}} \parallel v - v_A^{pref} \parallel_2$$

2.2 Problems with state-of-art methods

In this section, we present some of the most common problems among state-of-the-art methods.

2.2.1 Oscillation

Oscillation is one of the most well-known problems among velocity obstacle-based algorithms. It is mostly present in the standard *Velocity osbtacle* algorithm.

This behaviour happens in the scenario, when two or more agents are facing each other and heading the opposite direction. Once the agent detects a collision between them, none of them expects the other one to cooperate. Therefore, both customise their velocities to face away from each other. In the next step, they detect that they can both return to their previous velocities as they are closer to their preferred ones, returning them to the colliding state. We refer to this behaviour as an oscillation, and it can be seen in Figure 2.7.

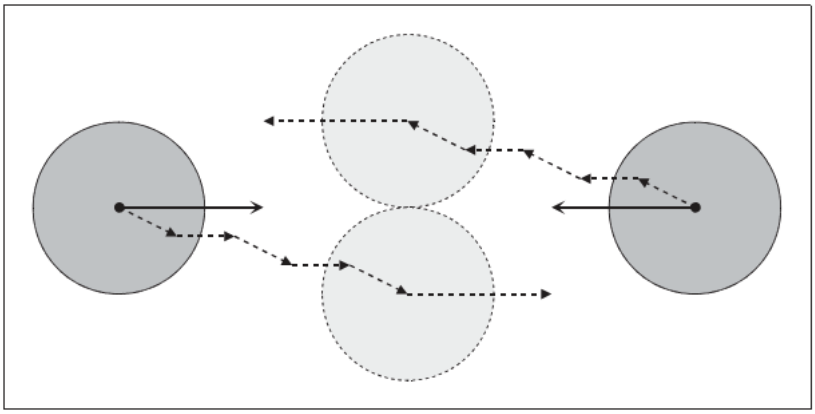


Figure 2.7 Visual representation of two agents oscillating. From *Reciprocal velocity obstacles for real-time multi-agent navigation*, by Berg, Lin and Manocha (17)

This problem was partially solved by the *RVO* method, where we expect the cooperation of other agents. We can still see the oscillating movement if the agents do not agree on the passing side. We refer to this as *reciprocal dances*.

2.2.2 Corner problem

This problem is specific to the *ORCA* algorithm. We take this example from *Game AI Pro 3: Collected Wisdom of Game AI Professionals*, by Steve Rabin (20). Usually, collision avoidance algorithms presume that agents have a straight path to the destination they should follow. This is often not the case in real-world scenarios, where agents often need to navigate around corners to reach their destination. During that corner navigation, agents typically change their preferred direction each frame. Consider an obstacle near the corner as shown in Figure 2.8. For agents to be able to successfully navigate around that corner, they will at some point need to change their passing side with that obstacle. Due to the nature of the *ORCA* algorithm, this change is not allowed. Therefore, agents often travel too far from their desired trajectory or simply become stuck in the corner.

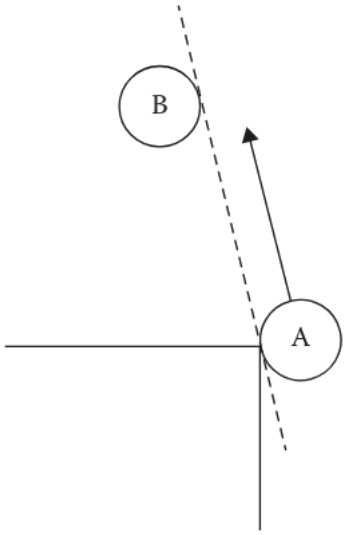


Figure 2.8 Visual representation of the corner problem for the *ORCA* algorithm. From *Game AI Pro 3: Collected Wisdom of Game AI Professionals*, by Rabin (20)

3 Genetic algorithms

In this chapter, we give a brief overview of Genetic algorithm (GA) and its applications. We start with a general overview and an example usage in Section 3.1 and then we explain how genetic algorithms can be used in real-time applications in Section 3.2. Lastly, we present a brief overview of the use of genetic algorithms in path navigation in Section 3.3.

3.1 General overview

Genetic algorithms are a subarea of a larger algorithmic group called *Evolutionary algorithms*. *Genetic algorithms* are widely used in the domain of global optimisation problems and local space search. They are inspired by biological evolution (hence the name), as they are using mechanisms such as mutation, recombination, and selection. We can see the steps of the genetic algorithm in Figure 3.1. We will go through each step and explain it in more detail in the following sections.

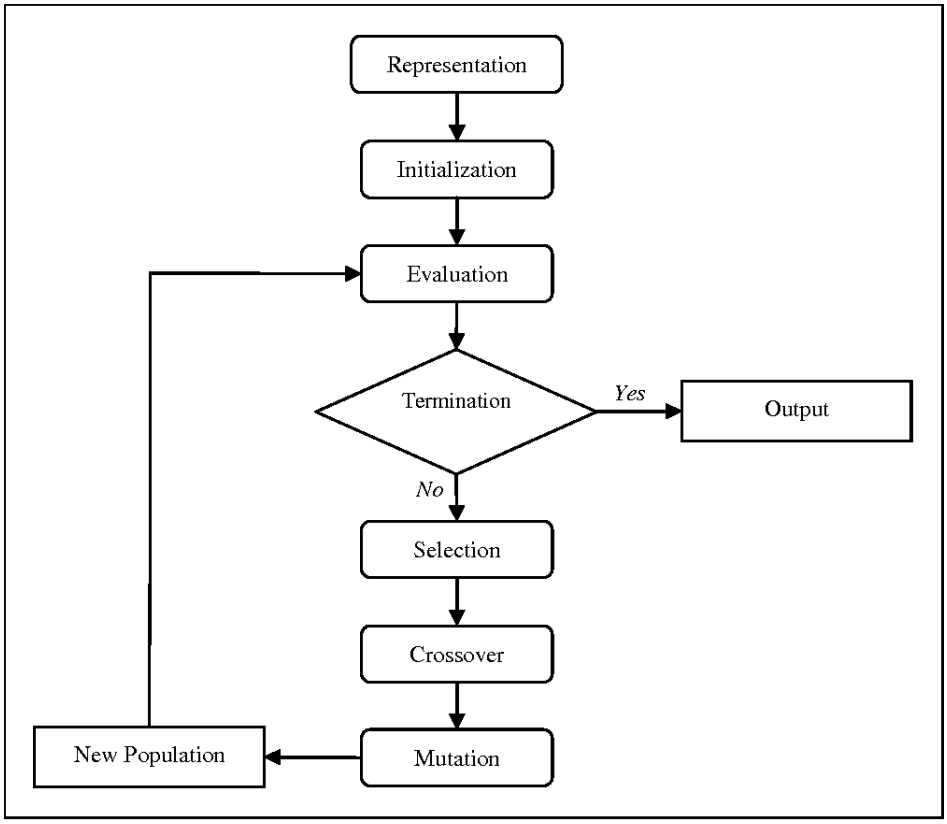


Figure 1. Steps of Genetic Algorithms

Figure 3.1 Steps of genetic algorithm. From *A Genetic Algorithm to Increase the Throughput of the Computational Grids*, by Reza Entezari-Maleki, Ali Movaghar (14)

3.1.1 Representation

The first step of each genetic algorithm is to identify the search space and create an individual representation.

In our case, the search space could be various paths that the agent can take to reach its destination. Our individual could then be represented as a single path, such that each path would consist of multiple segments of different sizes, which connect to each other under different angles. Individuals consist of the so-called *genoms*. In our path representation, one genome would correspond to one path segment.

Initialisation

Initialisation should happen once at the beginning of the algorithm and is used to create individuals. The group of individuals is called the *population*. Initialisation

can be random or more complete. If we have some constraints on the individuals, it is generally a good idea to adapt the initialisation method to incorporate them so that we create only valid individuals.

If we keep the example from previous Section 3.1.1 with individual being the agent's path, one constraint could be the length of given path.

Evaluation

The evaluation part is crucial from an overall evolution perspective. It is used to evaluate how good each individual in the population is. As a measurement, we use the property called *fitness* which is expressed by a numeric value, and the methods that calculate these fitness values are called *fitness functions*. In case of more complicated individuals, it is possible to have multiple fitness functions that evaluate different aspects of the individual. At the end of the evaluation, these fitness values usually need to be combined into a single fitness property. That can be done using various techniques, such as calculating the weighted sum of all fitness values.

Genetic algorithms usually aim at maximising the fitness values of individuals, meaning that individuals with higher fitness are considered to be better and are more likely to be selected into the next population. It is also possible to have fitness functions that aim at minimisation of the fitness value, but it needs to be consistent across the entire genetic algorithm.

Continuing on our example from the previous Section 3.1.1, we could have one fitness function evaluating how many collisions the agent encounters on its path. The other fitness function might evaluate the length of the path. Notice that in each of the examples we are aiming for minimisation; fewer collisions and shorter paths.

3.1.2 Selection

Selection is used to select individuals with leading fitness values and create a new population consisting of these individuals. This should ensure convergence towards better solution. There are multiple types of selection methods with different properties. Among the most common are, for example, *roulette wheel selection* and *elitist selection*.

Roulette wheel selection

In roulette wheel selection, we select an individual with a certain probability based on its fitness. This type of selection works if we maximise the fitness values of individuals and do not have negative fitness values. The probability for selecting

individual i is calculated as follows:

$$p_i = \frac{f_i}{\sum_{k=0}^{n} f_k},$$

where f_i is fitness value of individual i.

Elitist selection

Elitist selection is taking a predetermined number of individuals (less than the population size) that has the best fitness and moving them to the next population.

3.1.3 Crossover

Crossover, also sometimes called *recombination*, is one of two ways to create a new population. The crossover is a binary operator in genetic algorithms, that is, it takes two individuals (also referred to as *parents*) and create a single new individual. This operator is executed with predetermined probability. The selection of a concrete probability is then part of the tuning of the parameters of the genetic algorithm, which can lead to different results depending on the configuration. There are a couple of types of crossovers, based on how much information is taken from which parent. One of the most basic crossover types is the *uniform crossover*. This crossover is iterating over the parents' genoms and choosing one with equal probability.

The uniform crossover operator for our example from Section 3.1.1 could look like iterating over parents' segments, selecting the segment from one parent (with same probability for both parents) and connecting these segments into a newly created path.

Mutation

Mutation is the second method of creating a new population. It is usually a unary operator, which means that it takes one individual and modifies him into a new one. It can also create a completely new individual without any input. This operator is also executed with certain probability that can be modified depending on the configuration. Mutation is used to maintain diversity in our solution.

The mutation operator for our example from Section3.1.1 could look like iterating over individual's genoms (path segments) and randomly modifying their size.

3.2 Rolling horizon

As mentioned in this chapter, genetic algorithms are used for optimisation problems. Solving these problems usually takes a significant amount of time.

Therefore, it is less usual to use genetic algorithms in real-time applications, where we have time constraints on the maximum run time of the algorithm (which are usually tens or hundreds of milliseconds at most). We use "Rolling horizon evolutionary algorithms" approach presented by Perez, Samothrakis, Lucas, and Rohlfshagen as stated in the paper "an agent will evolve a plan in an imaginary model for some milliseconds, acts on the (real) world by performing the first action of its plan and then evolves a new plan repeatedly (again in a simulation based manner) until the game is over" (15) . This means that we let the algorithm run for small amount of time, collect results and apply them until the algorithm runs again. This happens repeatedly until the simulation is over.

If we map it to our agent's path example from Section 3.1.1, we would run the algorithm to obtain the agent's path and then navigate agent along that path. Navigation would happen until we run the algorithm again, which would give us another path that would replace the path from the previous run.

3.3 Related work

In this section, we review key concepts of some papers that tackle similar problems as agent's navigation and solve them using evolutionary algorithms.

3.3.1 EVOR: An Online Evolutionary Algorithm for Car Racing Games

First paper is *EVOR: An Online Evolutionary Algorithm for Car Racing Games*, by Nallaperuma, Neumann, Bonyadi, and Michalewicz (21). The idea behind this paper is to use an evolutionary algorithm to simulate a car by calculating its trajectory.

The idea is to train a controller that then navigates the vehicle. An individual in EA represents the acceleration, brake, and steering values. The individual consists of multiple heterogeneous genes with different ranges; therefore, standard crossover cannot be applied. As a mutation operator, the standard uniform mutation was selected. The algorithm runs indefinitely, receiving updates from sensors about a new car state, calculating new optimised state, and then converting results back to the values that are applied to the car.

Acceleration and brake in the individual are represented as a single float value in range -1 to 1, where positive numbers represent acceleration and negative values represent brakes. For steering, the floating point variable is used again, now in the continuous range -90 to 90.

The model of the track is represented as segments of lines perpendicular to the heading direction of the original track.

The fitness function is calculated using the track segments. For each track segment that intersects the calculated trajectory, the fitness function is incremented if their intersection point lies within the track boundaries. The fitness calculation also takes into account the current velocity of the car, checks for possible collisions with the track, and modifies the fitness accordingly. Fitness is also adjusted according to collisions with other cars. If the collision is detected with another car and that car's velocity is less than the car the algorithm is controlling, it is registered as a potential collision, and the fitness value is decreased.

The mutation for this algorithm is the standard uniform mutation. For the steering genom, it generates a random value from the continuous range -90 to 90. For the acceleration genom, it generates a random value from the continuous range -1 to 1.

3.3.2 Rolling Horizon Evolution versus Tree Search for Navigation in Single-Player Real-Time Games

Second paper is *Rolling Horizon Evolution versus Tree Search for Navigation in Single-Player Real-Time Games*, by Perez, Samothrakis, Lucas, and Rohlfshagen (15).

This paper compares various techniques such as *Monte Carlo Tree Seach* and *Evolutionary algorithm* in the problem known as *Physical Travelling Salesman Problem*.

The *Physical Travelling Salesman Problem* is a single player game where the player navigates the ship and needs to collect several waypoints scattered around the map full of obstacles. The player is supposed to collect all waypoints in as little time as possible. There are six actions for navigating the ship: acceleration (which can be either on or off), and steering (which is either left, right, or straight).

The evolutionary algorithm in this paper uses the rolling horizon approach presented in Section 3.2. It also uses the concept of the so-called *macro-actions*. As already mentioned, the player has six different actions available to navigate the ship. *Macro-action* is defined as the repetition of the same action for a predefined number of steps. Individuals are represented as a group of *macro-actions* with fixed length that is defined by the experimental setup. One population consists of 10 individuals that are randomly initialised. The algorithm uses the elitist approach, which means that it will promote the best two individuals to the next generation.

The mutation modifies only one of the inputs encoded in the individual (either acceleration or steering). Mutating the acceleration is performed by flipping its value from *on* to *off* or vice versa. The steering mutation is performed by changing the *left* or *right* values to the *straight* value, while mutation of the *straight* value is changed to the *left* or the *right* value with the same chance. The algorithm also runs

the tournament selection of size 3 to select individuals for the uniform crossover. There is also a second version of the algorithm in which neither tournament nor crossover operators are used.

Evaluation is done based on:

1. Distance and state (visited or unvisited) parameters of the next two waypoints in the route.
2. The time spent since the beginning of the game.
3. Collisions in the current step.

And the final value is calculated as the addition of following:

1. Distance points - we define reward for distance r_{dw} as $10000 - d_w$ where d_w is distance to the waypoint w. If the first waypoint is not visited, it is set to r_{d1}, otherwise it is set to $r_{d1} + 10000$.
2. Visited waypoints - the number of visited waypoints (of the next two), multiplied by a rewarding factor which is equal to 1000.
3. Time spent - set to 10000 minus the time spent during the game.
4. Collisions - penalisation of -100 if a collision is detected in this step.

4 Implementation overview

In this chapter, we go through the technical implementation of our solution. We focus mainly on the supporting architecture of our solution, which runs the whole program. A more in-depth explanation of the GA solution is described in the following chapter 5. We will mention some specific Unity-related terms, which explanation can be found in the Unity dictionary attachment A. First, we go through the technical overview of our application, introducing the reader to our engine choice and where the source codes are located (Section 4.1). Then we explain core simulation-related things such as fundamental classes present in simulation, game loop, agents' updates and its parallelisation, and scenarios management (Section 4.2). After that in Section 4.3, we give an overview of the most important interfaces and types in our solution. In the following Section 4.4 we look more closely into our agents' definition. In Section 4.5, we mention the usage of the third-party code in our solution.

4.1 Technical overview

To run the whole simulation, we decided to use Unity Engine (1). This is due to its availability, effective learning curve, and community support. We are using Unity version 2022.3.7f1. As a version control system, we decided to use modern state-of-the-art git, more specifically github. All source codes for this project can be found in Attachments C and are publicly available at `https://github.com/lakatop/UnityNavigation`. It is still an ongoing project, but for the purposes of this thesis, we use branch `BezierIndividual_only` and all the source codes described below are captured under the `v1.0.0` tag.

4.2 Simulation

In this section, we go through the core aspects of our simulation.

4.2.1 Class diagram

First, we show the simplified class diagram of the most important classes present in the simulation, which can be seen in Figure 4.1

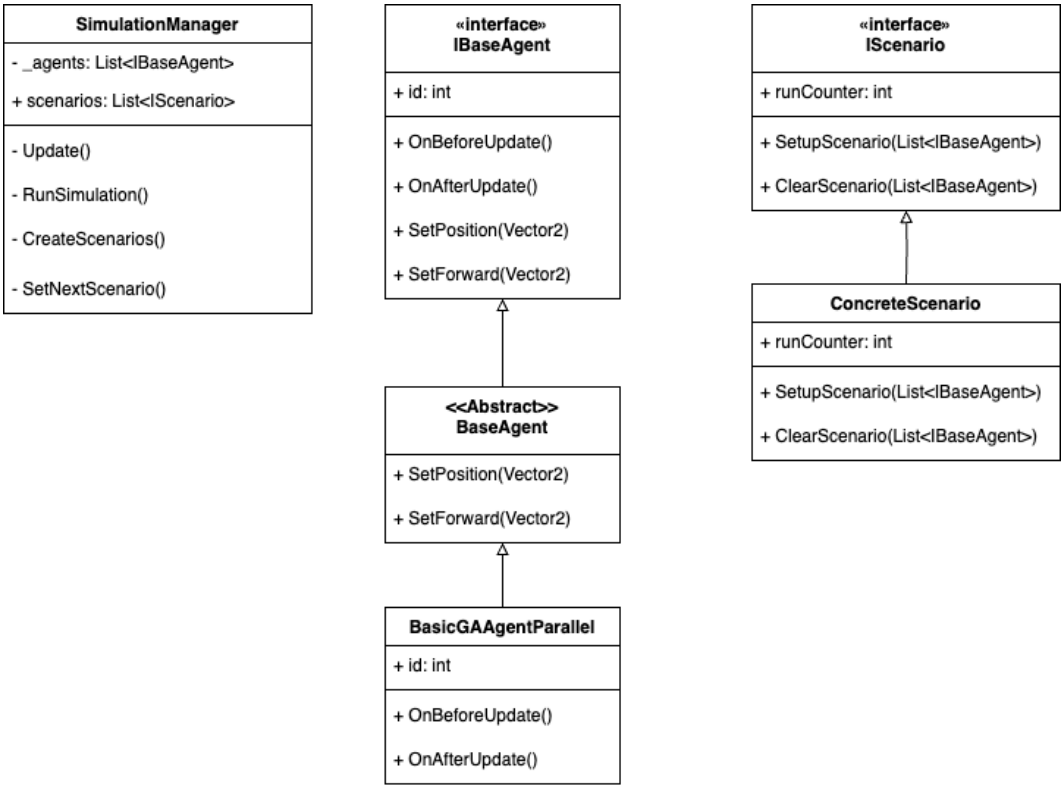


Figure 4.1 Class diagram of the most important classes in the simulation

The main class responsible for running the entire simulation is `SimulationManager`. Then, we have two main interfaces, `IBaseAgent` for agents and `IScenario` for scenarios. Then, we have abstract class `BaseAgent` for core definitions of agents methods, and derived class `BasicGAAgentParallel` for our concrete agent representation. From the `IScenario` interface we derive every concrete scenario. Each of these are described more closely later in this section.

4.2.2 Game loop definition

Unity as an engine has its own game loop defined. We use `SimulationManager` to define our own sub-game loop that is responsible for managing scenarios and updating agents. The sequence diagram of this game loop is visible in Figure 4.2. It is done inside the `Update` function that is called by Unity on each frame and has 2 main parts:

1. Managing scenarios - loading new scenario, checking whether scenario is finished (all agents arrive at their destination) and clearing resources from the previous scenario.

2. Managing objects inside the simulation (done inside the `RunSimulation` method) - managing resources related to the quadtree (more details described in Section 4.5), calling updates on agents (described in next Section 4.2.3).

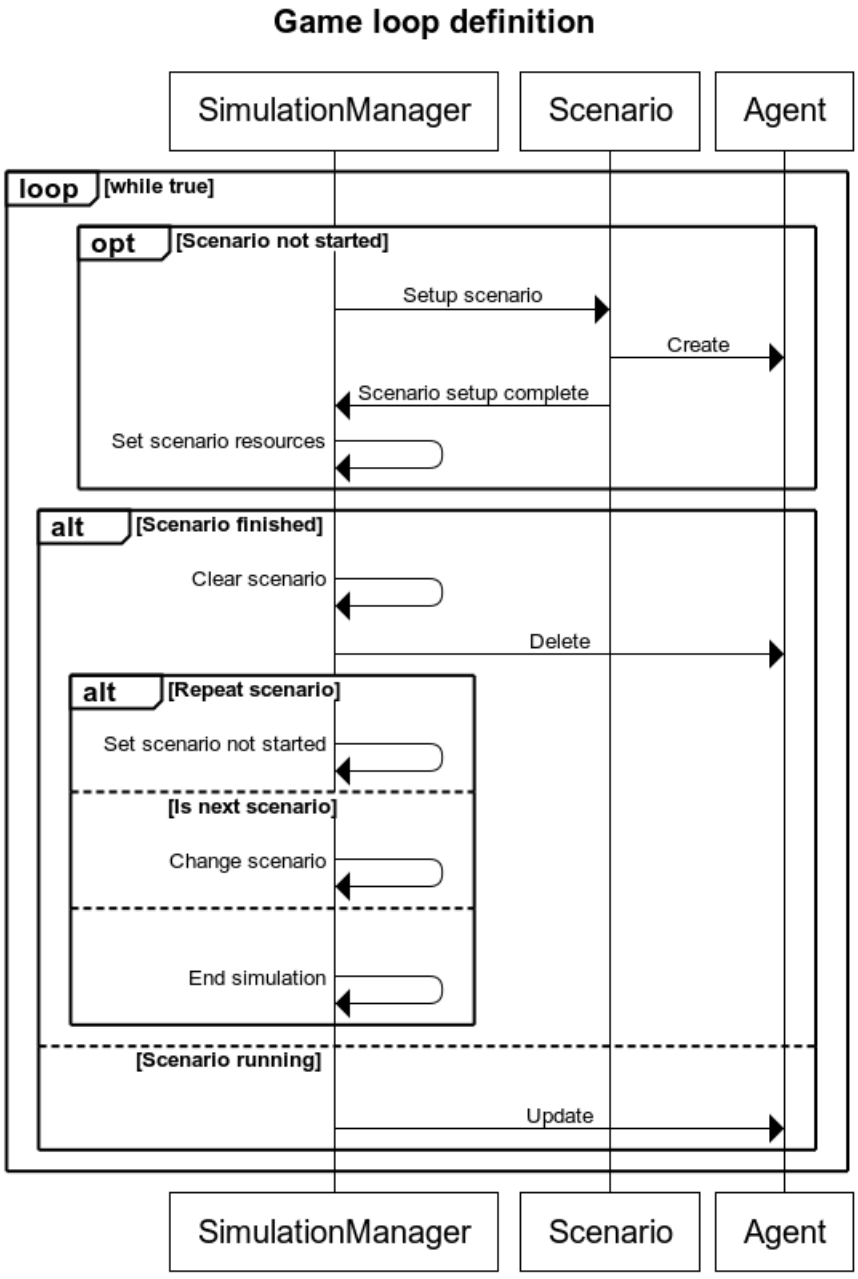


Figure 4.2 Game loop sequence diagram

4.2.3 Agents update

The `RunSimulation` method is also a place where we call updates on all agents present in the simulation. This is done for each frame using two methods `OnBeforeSimulation` and `OnAfterSimulation`. Currently, we are iterating the list of all our agents, calling `OnBeforeUpdate` and after that we again iterate over the same list and call `OnAfterUpdate`. These two methods are currently close to each other because we do not have anything else to update. However, the design of our solution is ready to be used in more complex cases, where you would first schedule the GA jobs inside the `OnBeforeUpdate` method (using parallelisation described in Section 4.2.4 below), then update the rest of the simulation, and after that apply the results in the `OnAfterSimulation` method.

4.2.4 Parallelisation

In this section, we describe the idea behind parallelisation and why we decided to use it. Unity has built-in parallelisation support called Job system (2). The workflow is as follows:

1. Create a C# struct that implements a `IJob` interface. This interface forces us to implement the `Execute` method which will be run when the job is scheduled.

2. Schedule a job - this is done by executing `Schedule` method on our struct. The `Schedule` method returns a `JobHandle`. The job is then scheduled to run on a separate thread.

3. We can optionally call `Complete` method on our obtained `JobHandle` from previous step. This will force Unity to finish the execution of the scheduled job and will not continue execution on the main thread until the job is done.

We already mention in Section 4.2.3 that we schedule a separate GA job for each agent in its `OnBeforeUpdate` function. During scheduling, we store `JobHandle` of a given job in the `_gaJobHandle` data member. Later in the `OnAfterUpdate` we call the `Complete` method on a `_gaJobHandle` to ensure that our GA algorithm will stop execution before reading the results.

In addition to this, we also used Unity's Burst compiler (3), which is an improved compiler to produce highly optimised native code using LLVM. This provided us with a significant performance boost. Without parallelisation, we would have to wait for each agent to sequentially run its GA algorithm and block the main thread. Using a parallelisation approach significantly improved our performance. This allowed us to run our genetic algorithm in multi-agent scenarios, whereas same scenarios without a parallelisation had noticeable performance problems.

Using Unity jobs has one downside from an implementation point of view. It allows us to use only Unity native collections (4) inside the struct methods that are executed. This means that we need to handle their memory allocation and deallocation (which is done by calling the `Dispose` method on given collection and is part of many interfaces described later in Section 4.3). It also means that in most cases we need to pre-allocate memory for collections that are used inside the GA.

4.2.5 Scenarios

Scenarios are classes used to define agents objectives. Each scenario in our solution has its own Unity scene (5). This scene contains following:

- Walkabe platform - a cuboid with a `NavMeshSurface` component with `Walkable` property. Each of these platforms have baked navmesh
- Static obstacles - cuboids with a `NavMeshModifier` component with `Not Walkable` property. Some of these obstacle may not be registered in the navmesh
- SimulationManager - empty object with `SimulationManager` script attached.
- Camera and lightning - Unity's default camera and light

The last thing that needs to be present in the simulation are agents. Each scenario class is responsible for populating the SimulationManager's `_agents` list with appropriate objects and setting them up. This should be done in the `SetupScenario` method.

`SetupScenario`

This methods takes reference to *SimulationManager's* `_agents` list so it can modify it and changes will be reflected in *SimulationManager* class as well. It populates the list with the appropriate objects. All our scenarios use `BasicGAAgentParallel`. Then it sets their destination and forward vectors to point in the direction of that destination. Lastly, it setups the logger class of a given agent with the following:

- Agent id
- Scenario name
- Start time - for this we use Unity's `Time.realtimeSinceStartupAsDouble` (6)

After the scenario is finished, *SimulationManager* is responsible for calling `ClearScenario` method. This method is used to set additional detail to agents' loggers (the end time of scenario), to create a corresponding csv file if it does not exist already, and log the results.

More about concrete scenarios and their purposes is described in Section 6.1.

4.2.6 SimulationManager

The main class responsible for running the entire simulation is `SimulationManager`. Its main responsibilities are the following:

1. Holding all relevant data to the simulation.
2. Defining our game loop (described previously in Section 4.2.2).
3. Managing agents in the simulation (described earlier in Section 4.2.3).
4. Loading of scenarios.
5. Transforming scenario objects into a new form that is representable inside `NativeQuadTree` that is explained in more detail in Section 4.5). This process is explained in Section 5.5.5.

Data relevant to simulation

The `SimulationManager` class is implemented as a singleton, therefore it can store data persistently during the whole simulation. The most significant data that this class holds are the following:

1. `_agents` - list of all agents present in the simulation.
2. `_obstacles` - list of all obstacles present in the current scenario (more in depth explanation of scenarios is in previous Section 4.2.5).
3. `_scenarios` - list of all the scenarios that will be simulated. We describe scenarios more closely in previous Section 4.2.5.
4. `_platform` - `AABB2D` bounds representing walkable platform in the current scenario. We explain `AABB2D` bounds in Section 4.5.
5. `_quadTree` - represents quadtree that we use to represent objects in space (more in Section 4.5).
6. `agentUpdateInterval` - determines how often agents run a new GA.
7. `_updateTimer` - temporary `Time.deltaTime` accumulator.

Scenarios management

Each frame we check the status of our current scenario, to see if we should continue updating agents or stop the scenario. We know that the scenario is finished once all agents are at their destination. If the scenario has not started yet, we need to set its resources by doing the following:

1. Creating agents and setting their destination.
2. Registering a static obstacles present in a scenario - these are the objects with `NavMeshModifier` component set to `Not Walkable`.
3. Registering walkable platform - it is the object with `NavMeshSurface` component. This is done to set bounds to our `_quadTree`.
4. Transforming obstacles to objects representable inside `_quadTree` (more described in Section 4.5).

Clearing the scenario then consists of the following steps:

1. Calling `ClearScenario` method defined by `IScenario` interface (more closely described in next Section 4.3).
2. Destroying all agents objects present in the simulation and clearing the `_agents` list.
3. Clearing the remaining resources related to obstacles and their `_quadTree` representation, as well as disposing `_quadTree` itself.

4.3 Interfaces and types

In this section, we go through the interfaces and most important structures present in our solution and describe their key parts. We can divide them into three following categories:

1. Agent related
2. Genetic algorithm related
3. Scenario related

4.3.1 Agent

Agent-related interfaces define actors and actor-related behaviours in our simulations. The main interface is `IBaseAgent` and the most important parts of it are the following:

1. `OnBeforeUpdate` - This method is called every simulation frame. It is a place where the agent should do the preparation before there is an update on the rest of the simulation. We use it for scheduling the GA job that will run concurrently on the other thread (more details are given in Section 4.2.4).

2. `OnAfterUpdate` - This method is called every simulation frame at the end of our game loop defined in `SimulationManager`. It is a place where the agent should update its position. We use it to stop the GA job triggered in `OnBeforeUpdate` and apply the results.

3. `SetPosition` - This method should be used to set agent's position.

4. `SetForward` - This method should be used to set agent's forward vector.

5. `id` - This property should be used to set or get agent's unique identifier.

6. `pathPlanningAlg` - This property is used to run the path planning algorithm provided by the built-in `NavMeshAgent`.

4.3.2 Genetic algorithm

First, we need to define structures that represent the individual and the population, which is crucial for every genetic algorithm.

Structures

The first structure we describe represents the GA individual - `BezierIndividualStruct` and has the following fields:

1. `Initialize` - method used for initialisation and resource allocation.

2. `Dispose` - method used for clearing the resources.

3. `fitness - float` data member that represents the overall fitness of the individual.

4. `bezierCurve - BezierCurve` data member that represents the individual's path as a cubic Bezier curve.

5. `accelerations` - `float` list that represents accelerations/decelerations of the individual on its path.

More in depth explanation on how we use given properties of the individual is in Section 5.5.1. As already mentioned, we use our internal structure `BezierCurve` to represent the cubic Bezier curve. It has the following fields:

1. `points` - `Vector2` list representing control points of the curve.
2. `Initialize` - method used to initialise resources.
3. `CreateInitialPath` - method used to define bezier curve by setting its control points.
4. `EvaluateQuadratic` - helper method used to evaluate the quadtratic bezier curve.
5. `EvaluateCubic` - method used to evaluate the cubic bezier curve.
6. `Dispose` - method used to clear resources.

The next important structure that we present is to represent the population. It is a simple structure called `NativeBezierPopulation` with the following fields:

1. `population` - list of `BezierIndividualStruct` representing given population.
2. `Dispose` - method used to clear resources.

Interfaces

All of the interfaces described in this section are generic, and are designed to be used with a structure that represents the GA individual as a parameter. The first interface that we define in this category is the `IParallelPopulation<T>`, which defines the interface for the population. It has an important method that we need from every population, and that is `Dispose`, which should be used to clear resources.

One of the most important interfaces is the `IParallelPopulationModifier`. We use this to define interface for all operators related to the GA (e.g. initialisation, selection, mutation, crossover, fitness). It contains the following methods:

1. `ModifyPopulation` - This method takes reference to an array of individuals and modifies the array accordingly. For example, for initialisation, we use it to iterate over dummy individuals and modify their parameters to form a proper population (we cannot allocate resources on spot because of parallelisation restrictions described in 4.2.4).

2. `GetComponentName` - This method should return the name of the component that implements it. It is used for logging the configuration of the whole GA.

3. `Dispose` - This method should be used to clear any resources related to the component that implements it.

Lastly, we have the `IGeneticAlgorithmParallel` interface used for the genetic algorithm. It has the following fields:

1. `iterations` - property that should defined how many iterations the GA should preform.

2. `populationSize` - property that should define how big is the population that the GA will run on.

3. `SetResources` - Method that can be used for setting additional resources to the algorithm.

4. `GetResults` - Method that can be used for returning the resulting velocity of the GA.

Because this interface is designed purely for the GA that will run in parallel using Unity jobs, we do not need any additional method in the interface for defining where the algorithm itself should run, because it is a good practice to run it inside the `Execute` method (described in Section 4.2.4).

4.3.3 Scenario

For scenarios we use the `IScenario` interface with the following fields:

1. `SetupScenario` - method that should be used to create agents present in the scenario and set their destination.

2. `ClearScenario` - method that should be called when the scenario ended. The scenario should do its last logic and cleanup in this function. We use it to log the results of a given scenario.

3. `runCounter` - property defining how many times given scenario should run before simulation will move on to the next scenario.

4.4 Agents

Our solution currently uses the `BasicGAAgentParallel` class that is derived from the main abstract class `BaseAgent` that implements the `IBaseAgent` interface (the most important parts of the agent's interface is described in previous Section 4.3.1). The two main functions of each agents are `OnBeforeUpdate` and `OnAfterUpdate`.

`OnBeforeUpdate`

This method is used to check whether the agent reached the destination and scheduling the new collision avoidance algorithm.

As the first thing in this function, we update the `_updateTimer` variable by `Time.deltaTime`. We use the `_updateTimer` to keep track of how often we run the GA. We then check whether we arrived at the destination and, if so, we set the agent's velocity to zero. If it is time to run GA, we first determine our destination (it can change because of the destination skipping feature that is explained later in Section 5.1) by calling the `CheckToSkipDestination` method and then we use the parallelism (described in Section 4.2.4), to schedule a new GA job.

`OnAfterUpdate`

This method is used to stop the running GA job (if there is any) and apply results.

Firstly, we check whether there is a job scheduled. If so, we know that the GA was (or maybe still is) running, so we order it to stop and obtain the resulting velocity (we refer to this as `nextVel`). We then calculate the next position of the agent as `nextVel * Time.deltaTime` and the next forward vector and update the agent accordingly. Lastly, we check whether we are in a range of setting the next destination from our path planning algorithm.

Path planning

Path planning is one of the two main parts of navigating the agent to the destination (we describe this more closely in Section 5.1). To execute the path planning algorithm, we use Unity's built-in component `NavMeshAgent` that is attached to Unity's object representing our agent. For this to work, each of our scenarios (more described in Section 4.2.5) consists of a walkable platform that has the baked in navmesh. We run the `CalculatePath` method that creates a `path` object holding the list of inter-destinations called `corners` (for more details refer to Section 5.1).

4.5 Third party code

As we described earlier in Section 4.2.4, we use Unity jobs for executing our genetic algorithm. As also mentioned in previous Section 4.2, we use a *quadtree* data structure to store objects' positions in the space. These positions are stored as points in a 2D plane. To avoid implementing the entire data structure with its functionality in Unity native collections, we used an existing implementation that is available in the NativeQuadtree repository on GitHub (7). From the provided implementation, we use the following:

1. `NativeQuadTree<T>` - generic quadtree data structure. We use our internal `TreeNode` structure (described later in this section) as a type parameter.

2. `AABB2D` - shortcut for axis-aligned bounding box represents a structure that defines one rectangle in space. Implementation comes with simple `Contains` and `Intersects` functions that test whether one `AABB2D` box is fully contained in another or whether it intersects, respectively.

To incorporate the `NativeQuadTree` into our solution, we defined `TreeNode` struct. This struct represents object stored in our quadtree and has the following properties:

1. `staticObstacle` - boolean flag whether object is static obstacle or whether it repesents an agent.

2. `agentIndex` - field is valid only if the given object represents an agent. Stores an agent's id.

3. `stepIndex` - field is valid only if the given object represents an agent. Stores the ordinal number of the agent's step (explained in more detail in Section 5.5.5).

As an initial bounds for our quadtree, we use our walkable platform object in the scenario. In this way, we ensure that we can represent the position of each of our objects inside the simulation.

5 Solution overview

In this chapter, we review the concepts of our solution. This is done from the perspective of genetic algorithms. Before that, we need to explain the main properties of the agent in Section 5.1. In the same section, we explain the basics of agent navigation. This is done to provide context to the reader. We then focus more closely on the specific parts of the basic generic algorithm in Section 5.2. After that, we introduce the initial implementation of our solution and then briefly sum up various improvements that we tried in Section 5.3 and Section 5.4 respectively. Lastly, we focus more deeply on the current solution that had the best results (Section 5.5).

5.1 Agent overview

Various parts of our agent's representation are closely linked to the choice of engine. Since we use Unity, our measurements are tied to its units. According to the documentation (12) 1 Unity unit is equal to 1 meter. We refer to these units as f or *float* units.

In our simulation, we represent the agent's body as a capsule because we are in a 3D space, but our GA is designed for a 2D space. Therefore in algorithm, we represent our agent as a circle on a 2D plane with the following properties:

1. Agent has a diameter equal to 0.5f.
2. Agent has a forward vector describing the direction in which the agent is heading.
3. Agent has maximum speed of $5f/s$, meaning he can travel 5.0 float units per second at most.
4. Agent has acceleration/deceleration of $2f/s$, meaning agent can change its speed (either up or down) by 2 float units per second.
5. Agent is configured to have paths calculated by the GA to be of the maximum size of 17.5f and have at most 7 segments (more in depth description of the agent's path and its segments is in section 5.5.1).
6. Agent has its destination that he is trying to reach.

A visual representation of our agent is shown in Figure 5.1. We can also see the red arrow connected to the body of the agent capsule. This arrow represents the agent's forward vector.

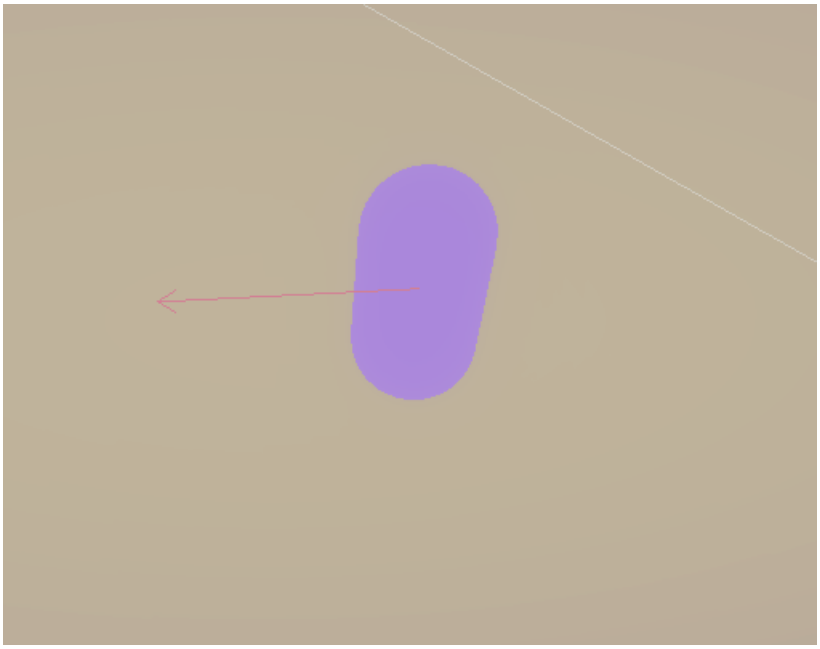


Figure 5.1 Visual representation of agent's body and its forward vector inside Unity

Navigating the agent towards its destination consists of two parts, *Path Planning* and *Path following*. To give a better understanding of how navigation works, we present the basic pseudo-algorithm of the game loop that demonstrates how many applications implement this behaviour. It can be seen in Algorithm 1.

Algorithm 1 Basic gameloop

```
1: function GameLoop
2:     while true do                                              ▷ Endless loop
3:         for all agent in agentsList do          ▷ Iterate over agents in simulation
4:             if agent.destination changed then
5:                 agent.CalculatePath()                    ▷ Perform path planning
6:             end if
7:
8:             if agent.position == agent.destination then
9:                 continue                                 ▷ Agent is in its destination
10:            end if
11:
12:            agent.CalculateNewVelocity()                              ▷ Path following
13:            agent.UpdatePosition()    ▷ Update agents position by new velocity
14:            agent.UpdateForward()           ▷ Update agents heading direction
15:        end for
16:    end while
17: end function
```

As described in Algorithm 1, basic game loop iterates over all agents present in the simulation and updates them. It detects whether the agent has a new destination set and, if so, it recalculates the agent's path (lines 4-6) by applying *Path planning* algorithm. It then checks whether the agent is at its destination, and, if so, moves on to updating the next agent (lines 8-10). After all checks have been completed, it starts working on updating the agent's position. First, the new velocity that will be applied is calculated (line 12). This part is usually some sort of *collision avoidance* algorithm, which calculates the most appropriate velocity that should be applied in the next steps, taking into account various predefined metrics such as speed or number of collisions. This can be solved using modern state-of-the-art methods such as RVO or ORCA (which were explained in Chapter 2), or we can apply our genetic algorithm (which is explained later in Section 5.5). After the calculation of the new velocity, the result is applied to the agent, which includes updating its velocity and heading (lines 13-14).

Path planning

In this section we explain in more detail how our solution performs *path planning*. Path planning is done to plan the path that avoids the obstacles marked in the navmesh. This might result in numerous inter-destinations captured in a list called *corners*. The agent's final destination is then the last element of mentioned *corners*

list. This path planning algorithm runs once when we set the destination to the agent. During navigation, we try to navigate the agent gradually through all of the inter-destinations until we reach the end of the *corners* list. During navigation, we implement something we call *destination skipping*.

Destination skipping Navigating gradually through inter-destinations has its disadvantages. The first main problem is that the inter-destination might get blocked by some dynamic obstacle that was not there during the path planning, resulting in the agent not being able to complete its entire path, even though in reality it cannot only reach one of the inter-destinations. The second problem is that our solution is designed to navigate the agent so that its velocity at destination is zero, which is not really necessary when we want to continue moving along the path. That is why we implement the *Destination skipping*. What it eventually does is to allow the agent to skip inter-destination and navigate towards the following destination in the *corners* list. The agent has defined a property called `maxSkipDestinations` which defines how many inter-destinations can the agent skip. Skipping is performed if `maxSkipDestinations` is greater than zero, and the agent is closer to the inter-destination than the agent's maximum path length. This should reasonably prevent the above-mentioned problems.

Collision avoidance

To perform collision avoidance, we run the genetic algorithm described in Section 5.5. The result of this GA is the velocity that is then applied to the agent.

5.2 Basic genetic algorithm

In this section, we describe the workflow of a basic GA. This is done via a pseudo-algorithm (see Algorithm 2) and shows the whole process of how we create and modify population throughout the generations. The implementation of individual parts of this GA with additional details is described in Section 5.3. What we can see in Algorithm 2 is that first we initialise the population (line 2 in Algorithm 2). Then perform a predetermined number of iterations in which we calculate fitness, select the best individuals for the next population, and apply operators (lines 5-11 in Algorithm 2.) After all iterations, we calculate the fitness of the individual once again and then select the overall winner (lines 14-15 in Algorithm 2), which is the individual with the highest fitness.

Algorithm 2 Genetic algorithm

```
1: function EXECUTE
2:     popInitialisation(pop)                          ▷ Initialise population
3:
4:     for i = 0; i < iterations; i++ do
5:         fitness(pop)                                ▷ Calculate fitness
6:         selection(pop)                              ▷ Perform selection
7:
8:         // Apply crossover operator(s)
9:         cross(pop)                                  ▷ Crossover operator
10:        // Apply mutation operator(s)
11:        mutation(pop)                               ▷ Mutation operator
12:    end for
13:
14:    fitness(pop)                                    ▷ Calculate fitness
15:    SetWinner()                                     ▷ Set the overall winner of GA
16: end function
```

5.3 Initial implementation

This section concludes the initial setup of our GA including individual representation (5.3.1), population initialisation (5.3.2), operators (5.3.3), and fitness and selection functions (5.3.4).

The initial implementation was done in the form of the basic genetic algorithm described more closely in the previous Section 5.2.

5.3.1 Individual representation

General idea was to represent one viable agent's path as one individual. Each path is divided into segments. Each segment represents one step of the agent. This is captured in Figure 5.2. We understand one step as a path agent will follow before the next run of the GA. The end point of the first segment of the selected path represents the place where the agent will be before re-executing GA. By having multiple segments in each path, we are predicting the overall agent's movement multiple steps ahead. To represent a segment, we decided to use two components:

1. Size - length of the segment represented as a `float` value.
2. Rotation - relative rotation to the previous segment (or agent's forward vector in case of the first segment) in degrees represented as a `float` value.

The whole path can then be stored in a `List` data structure. In addition, each individual also contains its fitness value, also represented as `float`.

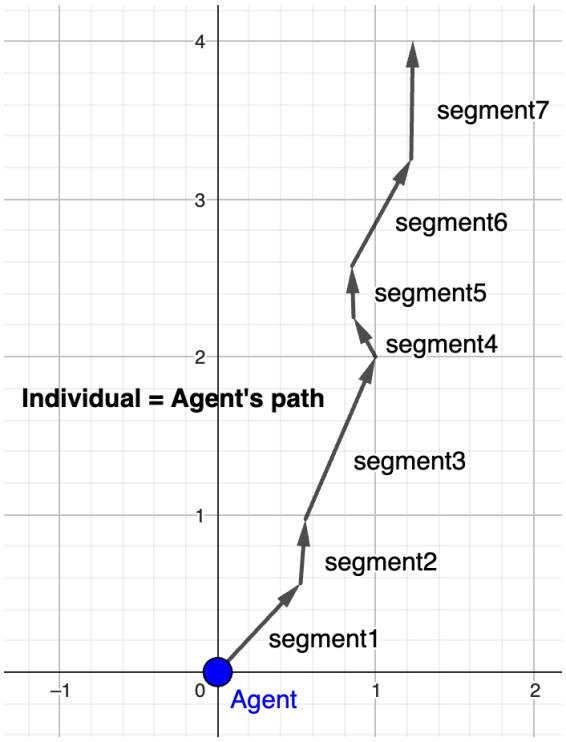


Figure 5.2 Visual representation of individual as an agent's path

The problem with this approach was that many paths were not heading toward the destination and the agents got off track. Another problem was that the paths were often unrealistic and had high jerk cost (which can be interpreted as the rate of change with respect to the object's acceleration over time), as we can see in Figure 5.3. The solution to this problem will be described in Section 5.5.1 as it fundamentally changes the GA.

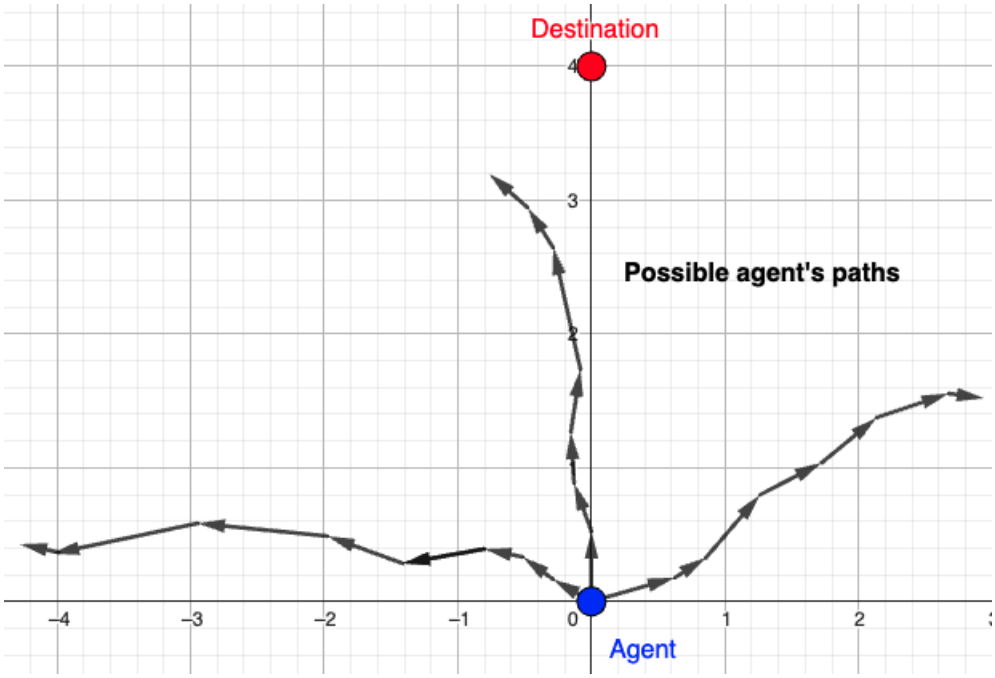


Figure 5.3 Example of possible agent's paths

5.3.2 Initialisation

Initialisation was random for each segment taking in consideration following constraints:

1. Size of one segment can be from 0 to `maxAgentSpeed * updateInterval`, where `maxAgentSpeed` is maximum agent's speed and `updateInterval` is interval how often the GA is run.

2. Rotation can range from -30 to 30 degrees.

5.3.3 Mutation and cross operators

For the crossover, we chose the uniform crossing for both size and rotation. Each crossover created two offsprings.

We also implemented a crossover that was calculating the mean of its parents. The operator was iterating over the population, selecting pairs of individuals next to each other, and creating the first offspring. The second offspring was created by randomly selecting one of the parents and one random individual from the population.

The mutation was to randomly change the size of the segment taking into account the initialisation constraints described in 5.3.2.

5.3.4 Fitness and selection functions

With fitness, we focused on the two most relevant metrics, the distance from the destination and the number of collisions. The distance from destination is calculated as the length of a straight line from the current position to the destination, without taking into consideration any obstacles. The resulting fitness was then calculated as $\frac{1}{\texttt{distanceToDestination}}$, or 0 if we collide in any of the path segments.
As a selection, we implemented basic roulette wheel selection.

5.4 Initial implementation improvements

In this section, we explain some improvements made to the initial implementation described in 5.3 to enhance its usability.

5.4.1 Initialisation

The basic initialisation described in 5.3.2 had two major problems. The first was that its starting rotation range was too low, making it difficult for agents to reach the destination that was not in front of them. The second problem was that any subsequent segment rotation was, on the other hand, too high, making the paths seem unrealistic, as we can see in Figure 5.4.

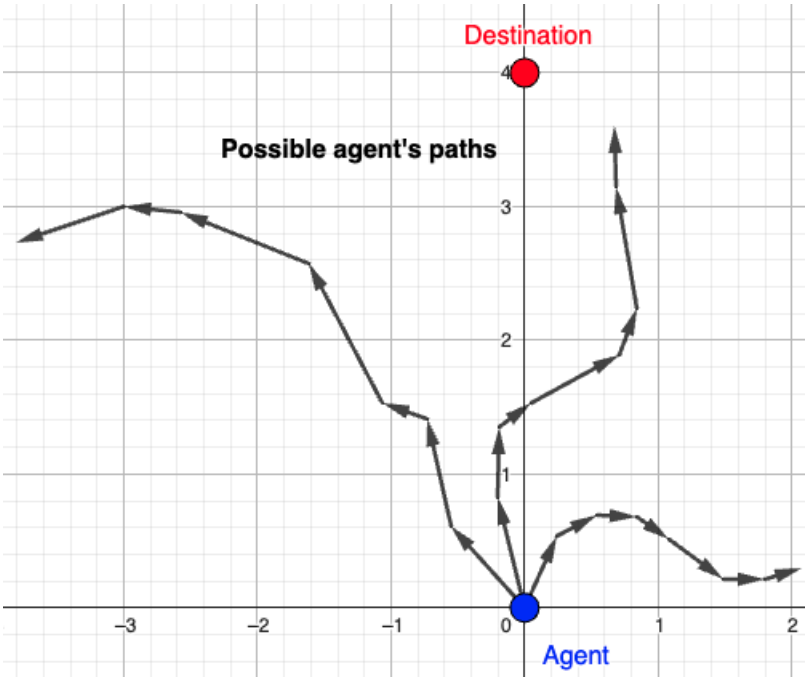


Figure 5.4 Example of possible agent's paths

To fix this, we introduced two new initializations:

Globe initialisation

To fix the problem of agents not being able to effectively change direction of their travel, we introduced the `Globe initialisation`. This initialisation took the first segment of each individual in the population and evenly distributed them in a 360-degree rotation so that each segment would have the same gap between them. Each subsequent rotation in the segment was at max 30 degrees. This fixed the problem of agents not being able to reach destinations that were not in front of them, but unfortunately introduced a new problem: agents were able to completely change direction after each run of the GA.

Kinetic-friendly initialisation

To fix the first problem that agents cannot reach the destination that is not in front of them, but also the problem introduced by the `Globe initialisation`, we introduce the `Kinetic-friendly initialisation`. This initialisation took the first segments of each individual, similar to the *Globe initialisation*, but its rotation range was reduced to 120 degrees. Each subsequent segment rotation was also limited to 15 degrees. This was our final initialisation in the current representation.

5.4.2 Mutation and cross operators

The problem with mutation until this point was that the mutation was purely random (selecting a random number from the appropriate range) and did not aim to improve the solution smartly. To introduce more smart and deterministic mutations, we added the following two:

Even circle mutation

This mutation detected that the agent is in reach of its destination and created a path from the current location of the agent to the destination. This was done either by straight line to the destination (if the difference between agent's forward rotation and straight line to destination was not bigger than 15 degrees) or by creating an arc of a circle that would evenly distribute velocities such that the agent would end up at its destination.

Greedy circle mutation

This was a modification of the previous *Even circle mutation* with one small change: the velocities in the circle arc were not evenly distributed, but used the

maximum agent's speed and then slowing before the destination.

These mutations were eventually dropped because of an unusual circular path that looked unnatural in the end.

5.4.3 Fitness functions

The problem with the previous fitness function was that it was too punishing when it comes to collisions. To better incorporate collisions and the overall distance of the segments from the agent's destination, we implemented an improved version of the basic fitness described in 5.3.4:

Continuous fitness

Initial fitness value was calculated as
`distanceToDestination`^{2} and then for each subsequent segment we calculated the position of the agent at the end of the segment and subtracted

- `distanceToDestination`$^2 * 2$ if agent is colliding
- `distanceToDestination`^{2} if agent is **not** colliding

5.5 Current solution

In this section, we describe the current solution that gives us the best results. First, we show the improved version of our GA in the form of a pseudo-algorithm (see Algorithm 3). All of the methods mentioned in this pseudo-algorithm are more closely described later in this section. All source codes related to these methods can be found in Attachments C in the *GeneticAlgorithm* folder.

Algorithm 3 Genetic algorithm

```
function EXECUTE
    popInitialisation(pop)                                  ▷ Initialise population

    for i = 0; i < iterations; i++ do
        jerkFitness(pop)                                    ▷ JerkCost fitness
        collisionFitness(pop)                               ▷ Collision fitness
        endDistanceFitness(pop)                             ▷ EndDistance fitness
        ttdFitness(pop)                                     ▷ TimeToDestination fitness

        ranking.CalculateRanking(
         jerkFitnesses, collisionFitnesses, endDistanceFitnesses, ttdFitnesses,
         jerkFitness.weight, collisionFitness.weight, endDistanceFitness.weight,
         ttdFitness.weight
        )                                                   ▷ Perform z-score normalisation and weighted sum
        ranking.WriteFitness(pop)                           ▷ Write resulting fitnesses to the individuals

        selection(pop)                                      ▷ Perform selection

        cross(pop)                                          ▷ Crossover operator
        controlPointsMutation(pop)                          ▷ ShuffleControlPoints mutation
        smoothMutation(pop)                                 ▷ SmoothsAcc mutation
        shuffleMutation(pop)                                ▷ ShuffleAcc mutation
        clampVelocityMutation(pop)                          ▷ ClampVelocity mutation
        straightFinishMutation(pop)                         ▷ StraightFinish mutation
    end for

    jerkFitness(pop)                                        ▷ JerkCost fitness
    collisionFitness(pop)                                   ▷ Collision fitness
    endDistanceFitness(pop)                                 ▷ EndDistance fitness
    ttdFitness(pop)                                         ▷ TimeToDestination fitness

    ranking.CalculateRanking(
        jerkFitnesses, collisionFitnesses, endDistanceFitnesses, ttdFitnesses,
        jerkFitness.weight, collisionFitness.weight, endDistanceFitness.weight,
        ttdFitness.weight
        )                                                   ▷ Perform z-score normalisation and weighted sum
    ranking.WriteFitness(pop)                               ▷ Write resulting fitnesses to the individuals

    SetWinner()                                             ▷ Set the overall winner of GA
end function
```

5.5.1 Individual representation

As already mentioned, the problem with the previous individual representation described in Section 5.3.2 was that many paths were not heading toward the destination and the agents got off track. Another problem was that the paths were often unrealistic and jerky. To fix all of the above problems, we decided to have a completely new representation of the individual. Each individual still represents a path, but now its encoded as a cubic Bezier curve starting from the agent's current position and ending at the agent's destination. We have chosen a Bezier curve representation because they are an industry standard in game development.

Another problem with representing segments as pairs of size and rotation was that it was easier to fall into situations that do not respect the kinetic laws. For example, we could get into a situation of rapid change in the agent's velocity going from its maximum speed to zero in one step. To prevent this behaviour, we defined the agent's maximum acceleration value, which describes how much a given agent can change its velocity in one step. Due to the Bezier curve, we no longer need the rotation for segments, since the path is already defined, and instead of size of the segment, we started keeping track of the agent's acceleration between the segments. It is encoded as a floating value in the range $[-1, 1]$ where -1 means deceleration at maximum capacity and 1 means acceleration at maximum capacity (concrete values are described in Section 5.1).

The resulting path is a piecewise linear approximation of the Bezier curve, where the size of the segment denotes the length of the curve between the start and end point of the given segment.

The final representation of the individual in code is described in more detail in Section 4.3.2

5.5.2 Initialisation

The implementation of initialisation was similar to our previous approach described in 5.4.1 in the sense of an even distribution. The cubic Bezier curve is defined by 4 points:

1. startPoint - initial/starting point of the curve
2. endPoint - final/ending point of the curve
3. C1 - first control point
4. C2 - second control point

The *startPoint* and the *endPoint* are well defined in each individual, because they are agent's current position and agent's destination respectively as mentioned in

5.5.1. The control points then describe the curvature of the spline. To create an even distribution among the population, we keep the same gap between each individuals' C1 points and C2 points, respectively. All C1 points then create one collinear group, and similarly, all C2 points create a second collinear group. We can see an example population of 50 individuals in figure 5.5, where green lines represents given individuals and purple capsule is the agent.

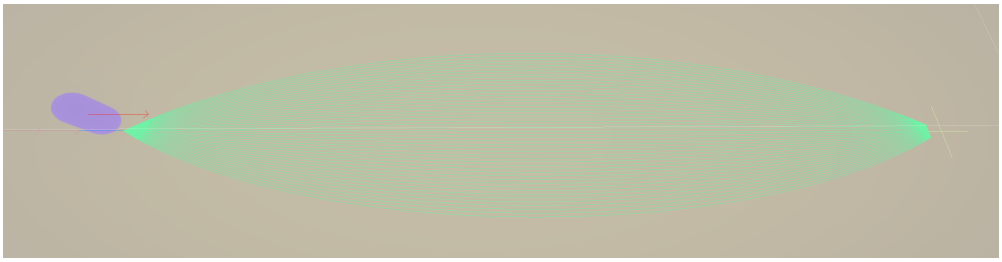


Figure 5.5 Initialisation example of popoulation of 50 individuals

To provide a reasonable rotation angle in the first segment, we use trigonometric functions to calculate the position of the control points. The position of the C1 point is more closely related to the agent's forward vector, while the position of the C2 point is determined on the $(startPoint - endPoint)$ vector.

Control points for each individual are created as follows:

- C1
    1. Place C1 point on the same position as startPoint
    2. Calculate $quadDistance = \frac{(endPosition - startPosition).magnitude}{4}$
    3. Translate C1 in direction of agent's forward vector multiplied by $quadDistance$
    4. Create a line perpendicular to the agent's forward vector. C1 will be placed on this line.
    5. Using trigonometric functions, calculate how much we can translate C1. We want to achieve that angle between the agent's forward vector and the $(C1 - startPoint)$ vector will not be greater than 30 degrees. In most scenarios, this should suffice to ensure that the agent cannot turn more than 30 degrees in a single step. Define the angle α that we want to have as the resulting angle. We can use the *tangent* function to calculate how much we can translate C1 on the perpendicular line as follows: $maxTranslateLength = \tan(\alpha) * quadDistance$. Now we can translate C1 into the perpendicular line by $maxTranslateLength$

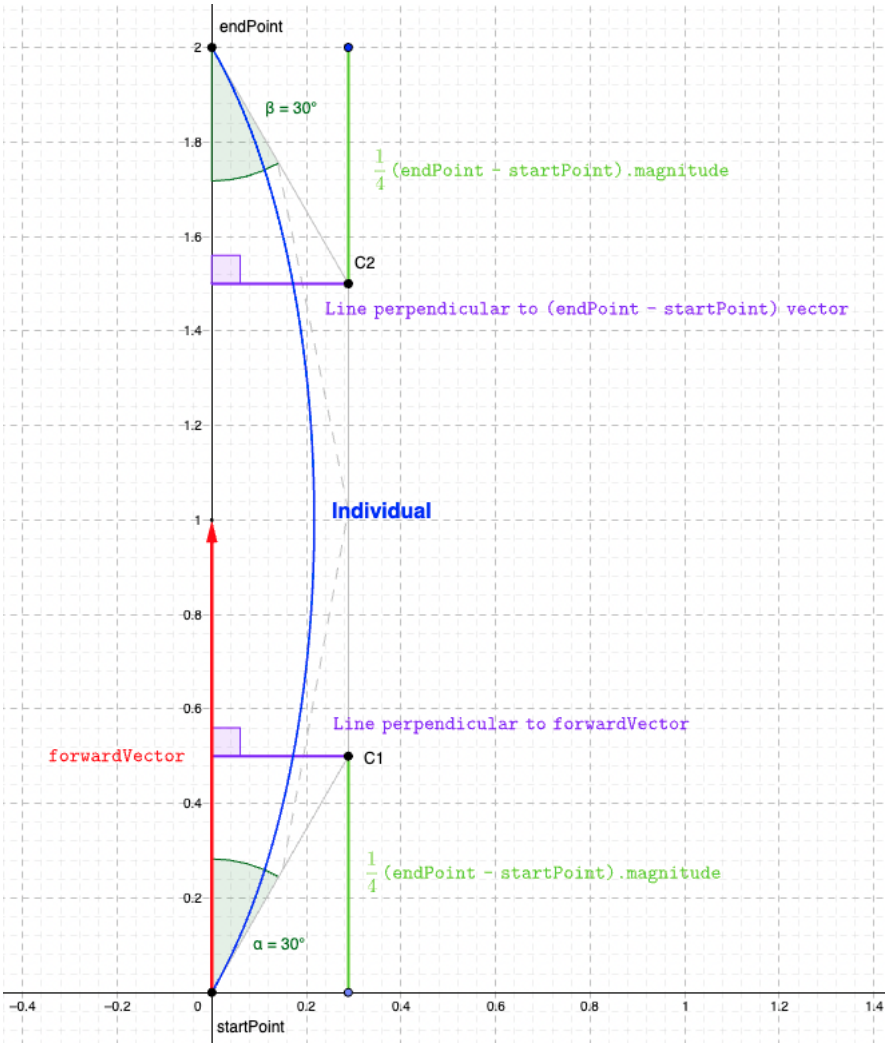


Figure 5.6 Bezier individual visualisation

- C2 - steps are analogous to the creation of C1, but instead of using the agent's forward vector, we use the $(startPoint - endPoint)$ vector.

For better understanding of the creation of control points described in 5.5.2, we can see the visual creation of the individual where $\alpha = 30°$ on the figure 5.6

We achieve an even distribution of individuals pictured in 5.5 by following steps:

1. First individual is always created with $\alpha = 30°$. We calculate $maxTranslateLength$ identically to the calculation described in 5.5.2.

2. We want to create individuals with identical curvatures on both sides, so we calculate $subFactor = \frac{(maxTranslateLength*2)}{populationSize}$.

3. We then iterate over the population and create individuals, but in each step we subtract $subFactor$ from $maxTranslateLength$.

We are aware of some edge cases where the constraint on the agent's turning angle is not met. One of the situations can be when an agent's forward vector is facing in the opposite direction to the agent's destination. Then it can happen that the overall segment size is so large in length that the resulting velocity vector has a rotation range greater than 30 degrees. These situations are currently neglected.

5.5.3 Mutation operators

In this section, we will describe the mutation operators that we implement. Before that, we need to describe the velocity clamping technique that is used by some of those operators.

Velocity clamping

This technique is used to determine the maximum velocity the agent can have by its default setting (described in Section 5.1) to be able to decelerate to 0 velocity at its destination. First, we need to understand that we can only change the velocity in discrete intervals every time an algorithm is run. By default settings, the GA is run each 0.5 seconds, the agent's maximum velocity is $5f/s$, and the agent can accelerate/decelerate $2f/s$ by each run of the GA. Since we are operating in the GA's intervals, we need to divide both velocity and acceleration/deceleration by half (because the GA is run twice a second). The example of deceleration can be the following:

- Agents current velocity in the GA is $2.1f/s$
- It takes 3 steps for him to decelerate to 0. If we want to use maximum deceleration for each step, it would look like the following:
    1. Agent's velocity is $2.1f/s$, he decreases his velocity to $1.1f/s$
    2. Agent's velocity is $1.1f/s$, he decreases his velocity to $0.1f/s$
    3. Agent's velocity is $0.1f/s$, he decreases his velocity to 0

If we want to determine the distance that agent will travel in the previous example 5.5.3, we would sum agents velocities:

$$2.1 + 1.1 + 0.1 = 3.3$$

We can create a general formula for calculating how much distance will agent travel if each step he decelerates by $1f/s$:

$$D = \frac{n}{2}[2a_0 - (n-1)]$$

where n is number of terms and a_0 is the first term. From this we can express a_0 as follows:

$$a_0 = \frac{D}{n} + \frac{n-1}{2}$$

The last thing we need to determine is the number of terms based on the velocity of the agent. This is equal to the number of steps that the agent needs to be able to decelerate to 0. We know that at the agent's maximum speed $2.5f/s$ it takes 3 steps and the overall distance of $4.5f$ to decelerate to 0. Therefore, with distances greater than $4.5f$ we can automatically assume that the agent can go at full speed and still be able to decelerate. Now we need to determine the steps for distances less than $4.5f$. These are presented in Figure 5.10:

Destination distance to deceleration steps table	
Destination distance	Deceleration steps
$[0-1]$	1
$(1-3]$	2
$(3-4.5]$	3

Figure 5.7 Table describing relation between destination distance and deceleration steps

StraightFinish mutation

This mutation aims to create a completely new individual with the following properties:

1. The Bezier curve has a shape of straight line directly to the agent's destination
2. In each step, it aims to use maximum possible velocity (calculated by the velocity clamping method explained in 5.5.3) agent can have at a given position

First, we check for the constraint of not exceeding the 30 degree rotation in a single step by comparing the angle between the agent's forward vector and the $(destination - agentPosition)$ vector. If this angle is greater than 30 degrees, we are not performing any mutation. Otherwise, we create a new individual with the properties mentioned above and replace last individual in the population by this newly created one.

ClampVelocity mutation

The idea of this mutation is to check the first segment in individual and clamp its velocity if it exceeds the maximum velocity (calculated by the velocity clamping method explained in 5.5.3) that the agent can have in its current position to still be able to decelerate to 0 velocity at the destination. The crucial part of this mutation is to calculate the path length, which is done by estimating the length of the Bezier curve and then using a piecewise linear approximation of the given curve to divide it into multiple segments and summing up the length of these segments. The estimation of the Bezier curve is calculated as follows: $estimatedLenght = \frac{|C1-startPoint|+|C2-C1|+|endPoint-C2|+|endPoint-startPoint|}{2}$

The number of segments is then calculated as

```
CeilToInt(estimatedLength * 10)
```

SmoothAcc mutation

This mutation aims to reduce the overall jerk of the path by sudden changes in the agent's accelerations/decelerations. It iterates through the individual's acceleration array, takes adjacent pairs, computes their average, and writes the result back to both of these acceleration units.

ShuffleAcc mutation

This mutation brings randomness into solutions, which is important for space search algorithms. It iterates over an individual's acceleration array and randomly changes its values (in the appropriate range, which is $[-1, 1]$ as described in 5.3.1).

ShuffleControlPoints mutation

This mutation brings another random element into the solutions, changing the positions of the control points (in an appropriate space, taking into consideration the conditions we defined in the Initialisation subsection 5.5.2). The workflow of mutation is as follows:

1. We calculate the $halfDistance = \frac{(endPosition-startPosition).magnitude}{2}$ which is equivalent of $quadDistance$ from initialisation section 5.5.2, but we increased the range from $\frac{1}{4}$ to $\frac{1}{2}$.

2. We randomly generate a number in the range of $[0 - halfDistace)$ that we call $upDistance$ - this is basically our new $quadDistance$.

3. Appropriate $maxTranslateLength$ (defined in initialisation section 5.5.2) is calculated based on $upDistance$

4. We randomly generate number in range $[-maxTranslateLength, maxTranslateLength)$ that we call $sideDistance$ – this is basically our new $maxTranslateLength$

5. We calculate new positions of both C1 and C2 based on these values and assign them to the selected individual

5.5.4 Cross operator

The overall solution relies on mutation operators rather than crossover. It does not make much sense to use many smart crossover operators, so we created one to again introduce some more variability and randomness into our solution. This crossover might create individuals who do not have control points on the same side, creating curvature that resembles the letter "S". To do so, we implement it as a uniform crossover operator for the control points' coordinates of both selected parents.

5.5.5 Fitness functions

In this section, we explain how we calculate the resulting fitness of each individual. One main difference from previous solutions described in 5.3.4 is that now we are using multiple fitness functions (each evaluating different aspects of the individual) rather than one complex fitness function. We also need to note that in our implementation of GA, we aim to minimise all our fitness functions. Using multiple fitness functions means that we need to combine these multiple fitness values into a single score. For that we use z-score normalisation and weighted sum, more closely described later in this section. Before we get into concrete fitness functions, we need to explain how we detect collisions of our individuals, since it is a key functionality in some of the fitness functions. We then also explain one of the core principles in every fitness function, which is dealing with an agent overshooting its distance.

Collision detection

To keep the positions of all objects present in the scenario, we use `NativeQuadTree` (explained in Section 4.5). This quadtree is shared among all agents in the simulation and is created from scratch every time the agents are about to perform their GAs. We also need to track agents' future positions. The design of this algorithm was meant to be independent of communication with other agents (because they might implement other collision avoidance algorithms or even be controlled by a human player); therefore, future positions are approximations. We calculate them as follows:

1. We take agent's current position, forward vector, velocity, and simulation's `updateInterval`.

2. We keep this metrics as they are and calculate where the agent will be in the following n steps, where n is equal to path segments in agent's configuration described in 5.1. These positions are then added to the quadtree with their corresponding `stepIndex` (field of the `TreeNode` struct explained in Section 4.5), which is the ordinal number of the step.

3. We fill in gaps between calculated positions (to be able to query them from the quadtree). We take into consideration the agent's radius and place points next to each other evenly spaced with the space size of $2 * agentRadius$.

To capture static obstacles (such as walls) in the quadtree, we represent them in the same way as agents. The idea is that each obstacle is the same as a group of static agents standing next to each other in a given area. We are filling these obstacles the same way as we were filling gaps between agents' steps. This is to capture the total number of collisions that would occur if agents try to cross the solid obstacle. The visual representation can be seen in Figure 5.8 below. The centre of the black circles represents the positions stored in the quadtree, and the circle radius is the same as the agent's radius.

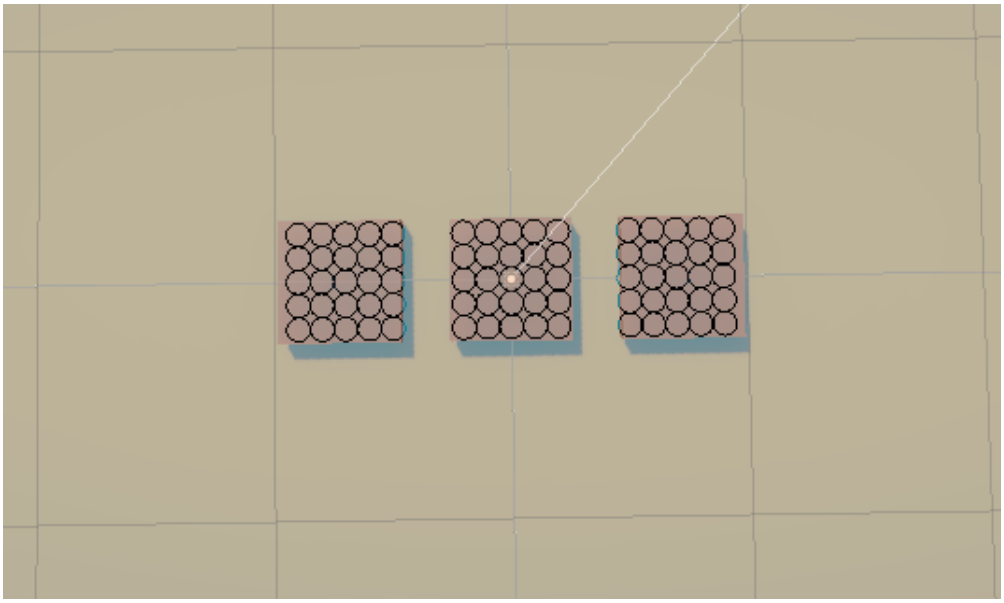


Figure 5.8 Static obstacle representation in quadtree (note: The circles in the lateral obstacles are not centred due to the camera angle)

The final collision detection is done in following steps:

1. Take current agents position and define bounding box for the quadtree query (we use $5fx5f$ sized box)

2. Retrieve all surrounding points from quadtree that are in bounding box with a center in current agent's position

3. Check all retrieved points whether they do not represent current agent. If yes, do not count it as a collision

4. Generally, if the distance between agent's current position and given point is less than $2 * agentRadius$, we know that these collide. But with other agents, we need to check the condition on `stepIndex` as well. This is necessary to do, because we may collide with other agent's position but in a different step - therefore, collision would not happen. If their `stepIndex` match, we count it as a collision.

Destination overshooting

With any path creation algorithm, there is always a risk of creating a path that will overshoot the destination, telling the agent to continue even after passing the destination mark. In our fitness functions, we incorporated penalisation for overshooting the distance. The reason is that if an agent exceeds its distance, it needs to decelerate, turn around, and return to the destination. The overshoot can be detected if we get to the destination (or close to it) and one of the two following is true:

1. Agent will not be able to decelerate to 0 when arriving to the destination

2. There are still some non-zero inputs in `accelerations` array that are telling agent to apply some other velocity than 0 that would eventually position him off target.

Due to the nature of our solution, we know that all paths are passing through the destination (because the Bezier curve's last point is destination). Therefore, if we detect an overshoot in our solution, the resulting vector agent applies is calculated as $(destination - currentPosition) * velocity$, where the velocity is calculated from the overshooting acceleration. Before counting it as a real overshoot, we first detect by our velocity clamping algorithm 5.5.3 whether the agent's velocity at a given moment is less than the maximum velocity for proper deceleration to 0. If true, we do not count it as an overshoot, because we are counting on several of our mutations to be able to clamp the velocity.

EndDistance fitness

This fitness function is responsible for evaluating the first metric that we defined in Section 1.2. Its main purpose is to evaluate how close the agent is to the destination. This is done by taking the agent's final position on his path and calculating the straight-line distance between that position and destination. We take this destination as a result of fitness. In case of overshoot, we do following:

1. Calculate distance to destination from agents position before overshooting (we refer to this as $distanceToDestination$)
2. Calculate the length of the path after passing the destination with agent decelerating by its maximum possible deceleration, we refer to that as $traveled$.
3. The resulting fitness is calculated as

$$distanceToDestination + (traveled * 2)$$

TimeToDestination fitness

This fitness function is responsible for evaluating the second metric defined in Section 1.2. Its main purpose is to track how many steps the agent takes to reach the destination. If the agent does not end at destination, the resulting fitness value is the length of its path. In case of overshooting the destination, we do the following:

1. We mark down how many steps agent already made and one more for crossing the destination. We refer to this as $pathSize$
2. Similar to what we did in `EndDistance` fitness when overshooting, we calculate how many steps it would take agent to stop completely after crossing the destination. We refer to this value as $afterDestinationPathSize$
3. The resulting fitness is calculated as

$$pathSize + (afterDestinationPathSize * 2)$$

Collision fitness

This fitness function is responsible for evaluating the third metric defined in Section 1.2. Its main purpose is to evaluate the path based on how many collisions it contains. To place greater emphasis on collisions that occur earlier in the path, we use the following decay function:

$$e^{-0.5*(stepIndex-0.5)}$$

where *stepIndex* is the ordinal number of the segment (and eventually refers to the same *stepIndex* that we used in the collision avoidance Section 5.5.5) This decay function can also be seen in Figure 5.9

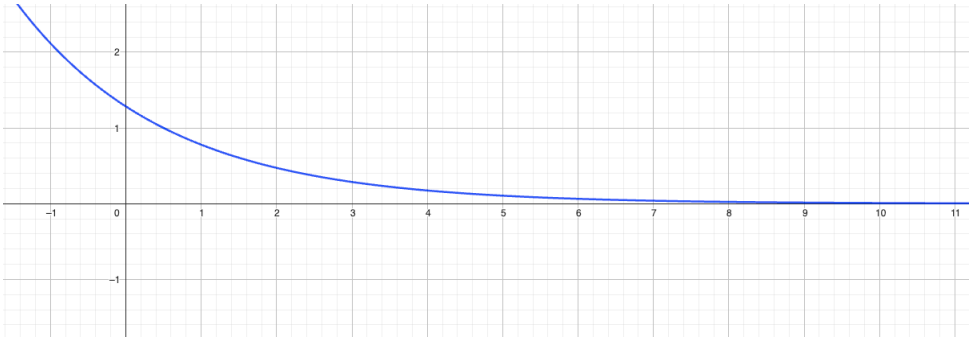


Figure 5.9 Collision decay function

We then iterate over each step of the agent's path, calculating the number of collisions in a given step and multiplying it by our decay function. The resulting fitness is then the sum of these values across the whole path.

JerkCost fitness

This fitness is responsible for evaluating the fourth metric defined in Section 1.2. Its main purpose is to evaluate the path based on its jerk value. In physics, the jerk is defined as the third derivative of positions. We can think of it as the rate of change with respect to the object's acceleration over time. To calculate the jerk value, we do the following:

1. First, we create an array representing agent's velocities. We do this by calculating the current agent's path segments (this is done by piecewise linear approximation of the Bezier curve as mentioned in 5.5.1). From that we get the agent's starting and ending position for each segment.

2. We iterate over positions from the first step, always taking the current position (we refer to this as *currentPosition*) and the previous position (we refer to this as *previousPosition*). We then subtract *previousPosition* from *currentPosition* which results in velocity vector that we store in separate array (we refer to this as *velocityArray*).

3. We iterate over *velocityArray* the same way we iterated over positions and again subtract current velocity from previous one. This results in acceleration vector which we store in another array (we refer to this as *accelerationArray*).

4. Finally, we iterate over *accelerationArray* and subtract previous acceleration from the current one, which gives us jerk that we again store in separate array (we refer to this as $jerkArray$).

5. We then calculate the sum of square magnitude of all jerk vectors, divide it by count of the jerk vectors, and take the square root of that value. This value is then our resulting fitness.

We can see that *velocityArray* is the same size as our individual's acceleration array (defined in Section 5.5.1). In case of overshoot, we similarly to `EndDistance` fitness do the following:

1. Calculate the agent's velocities in full deceleration until it reaches velocity 0.

2. Calculate the agent's velocities as he would return straight back to the destination in evenly distributed steps of size 1f.

3. Fill in the rest of *velocityArray* with calculated velocities (*velocityArray* is of fixed size, so not all of them might fit).

Z-score normalisation

All our previously calculated fitness values are in a different range of values. If we want to use a weighted sum described below, we need to normalise these values. To do so, we use the z-score normalisation. This is mainly because it does not require specific range boundaries, such as the minimum or maximum. We have four arrays of fitnesses holding fitness values for every individual, each for one fitness function. For each of these arrays, we do the following:

1. We start with an array of fitnesses, referred to as $fitnessValues$.

2. We calculate the average of values stored in $fitnessValues$, which we call $averageFitness$.

3. We calculate the *squaredSum* variable, which is done by iterating over all elements of $fitnessValues$ and calculating $squaredSum+ = (averageFitness - fitnessValues[i])^2$, where $fitnessValues[i]$ is ith element of $fitnessValues$ array.

4. We then calculate the $variance = \frac{squaredSum}{fitnessValues.Length}$.

5. Next, we calculate the standard deviation $stdDev = \sqrt{variance}$.

6. If $stdDev$ is equal to 0, all normalised values are 0, else we create a new array of normalised values $normF$ and fill it by iterating over $fitnessValues$ and calculating $normF[i] = \frac{fitnessValues[i] - average}{stdDev}$, where $normF[i]$ and $fitnessValues[i]$ are ith element in $normF$ and $fitnessValues$ arrays respectively.

This gives us four separate $normF$ arrays of normalised fitness values, each for one fitness function. We then use these $normF$ arrays later in the weighted sum described below in this section.

Weighted sum

As a result of the normalisation of the z-score (described earlier in this Section), we have four arrays of normalised fitness values, which contain fitness values for each agent. This means that we still have four separate fitness values for each agent. To be able to compare individuals between each other, we decided to use a weighted sum method, which will combine these four fitness values into one that we will use as the overall fitness of the individual. Weighted sum is a technique how to assign weight to each fitness value respectively. Each fitness function has a defined weight, how much should it affect the resulting fitness. Our baseline solution (defined in Section 6.2.1) has the following weights:

Fitness weights in the baseline solution	
Fitness function	Weight
Collision fitness	0.5
EndDistance fitness	0.2
JerkCost fitness	0.2
TimeToDestination fitness	0.1

Figure 5.10 Table describing weights of our fitness functions in the baseline solution

It is considered good practice to use weights such that their sum is equal to 1. The resulting fitness for each individual using the weighted sum is then calculated as follows

$$resultingFitness = normF1 * w1 + normF2 * w2 + normF3 * w3 + normF4 * w4$$

where $normF\{n\}$ is the normalised value of the nth fitness and $w\{n\}$ is the weight of the nth fitness.

5.5.6 Selection

As a resulting selection algorithm, we decided to use the elitist approach. This means that for each selection we decide on a number of individuals that will survive to the next generation. To do so, we do the following:

1. Choose number of individuals that will be taken to the next generation (we refer to this number as n), which is usually half of the population.

2. Sort individuals array in ascending order (we refer to this array as $individualsArray$).

3. We iterate over $individualArray$ and do the following $individualArray[i] = individualArray[i\%n]$, meaning we override individuals that are pass the limit by individuals starting from the beginning of the $individualArray$.

6 Evaluation

This chapter consists of three parts, design, results, and discussion. In the design Section 6.1 we go through the design of various scenarios thatwere implemented to test our implementation. In the same section, we also describe what hyperparameters were selected for the experiments in given scenarios. In the next Section 6.2 we present our baseline setup of hyperparameters, what sweeps across these hyper-parameters we performed, and what the results are. Lastly, we leave some space for discussion in Section 6.3.

6.1 Design

In this section, we first review the prepared scenarios that we use for the evaluation of our solution.

6.1.1 StraightLine

This scenario is used to evaluate how the agent handles the simple use case where the path should be a straight line to the destination. We expect the agent to go directly to the destination. The entire scene consists only of a walkable platform without obstacles. The agent starts at position [0,0] and its destination is set at [0,40]. We should note that in all scenarios, the starting forward vector of the agent is calculated as $(destination - startPosition).normalized$, which, in other words, means that the agent always heads in the direction of its destination. Unity's dedicated scene for this scenario in our solution is called *StraightLineScene*.

6.1.2 SmallObstacle

This scenario is used to test the use case when there is a small obstacle in the agent's way that is not registered in navmesh. The agent starts at position [0,0] and its destination is set at [0.40] as in the previous *StraightLineScene* scenario. The only difference is that, approximately in the middle between agent's start position and its destination, we placed a cube of size 3x3 units that is not registered by navmesh. This should test whether the agent will avoid collision with a static obstacle using our genetic algorithm. We can see the initial layout in Figure 6.1. Unity's dedicated scene for this scenario in our solution is called *SmallObstacleScene*.

Figure 6.1 Initial layout of the SmallObstacleScene

6.1.3 CornerSingle

In this scenario, the agent is forced to reach the destination by passing near the corner of the static obstacle. This obstacle is registered in navmesh, which means that the agent's interdestinations will be at the corners of the obstacle. We are trying to observe whether the agent avoids a typical problem in this use case, such as wall hugging. The agent starts at position [-40,20] and its destination is set to [-40,30] and between them there is a static obstacle which starts at the edge of a walkable platform, forcing the agent to bypass the obstacle from one side only. We can see the layout of the scene in Figure 6.2. Note that the agent is heading towards the destination at the start, so he first needs to turn in the direction of the corner, which will most likely put him in the position closer to the wall. Unity's dedicated scene for this scenario in our solution is called *CornerScene*.

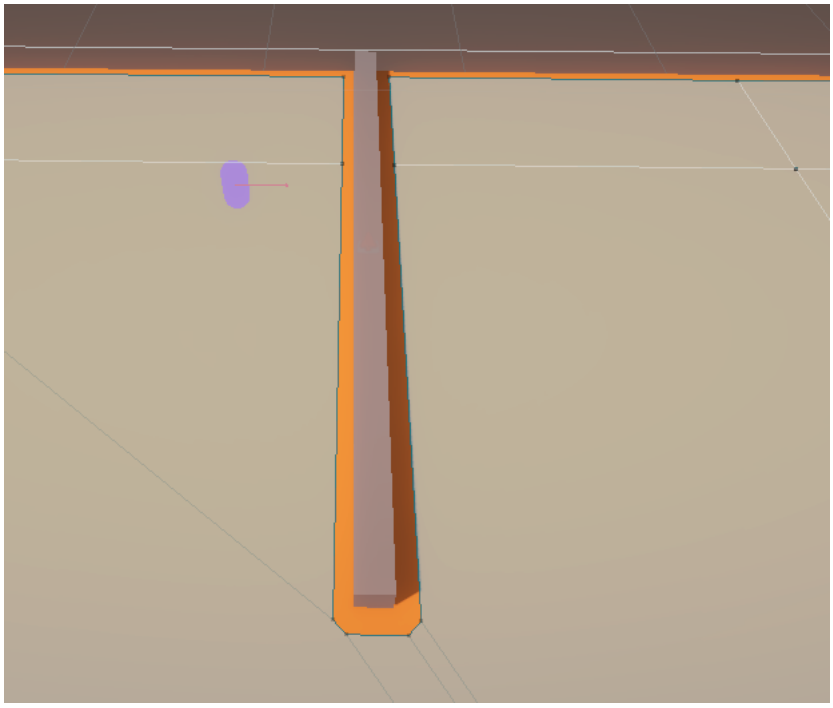


Figure 6.2 Initial layout of the CornerScene

6.1.4 OppositeAgents

This scenario is a classic example presented by multiple state-of-the-art collision avoidance algorithms. We have two agents that are 30 units (for units explanation, refer to Section 5.1) apart and facing each other. Each of the agents has a destination set on the opposite side. This results in agents moving towards each other to reach their destinations. The concrete coordinates for the first agent are the start position at [0, -20] and the destination at [0,30], whereas for the second agent it is the start position at [0,20] and the destination at [0,-30]. This scenario should test two important things, whether the agents avoid collision with each other and whether they will not start oscillating. Unity's dedicated scene for this scenario in our solution is called *OppositeScene*.

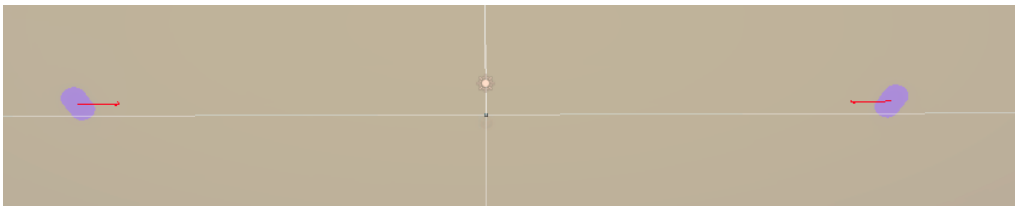


Figure 6.3 Initial layout of the OppositeScene

6.1.5 OppositeMultipleAgents

This scenario is basically the same as the *OpositeScene* scenario but on a larger scale. Instead of having two agents facing each other, we now have ten agents, five on each side. The agents are placed right next to each other without any additional space between them. Concrete coordinates can be found in the source codes (see Attachments C) inside the `Scenarios` folder in `OppositeMultipleScenario.cs`. This scenario should test for collisions and oscillations the same as the previous *OpositeScene* scenario, making it harder for agents because now they also have other agents moving next to them, as well as multiple agents opposite them. We can see the initial layout of the scenario in Figure 6.4. Unity's dedicated scene for this scenario in our solution is called *OppositeMultipleScene*.

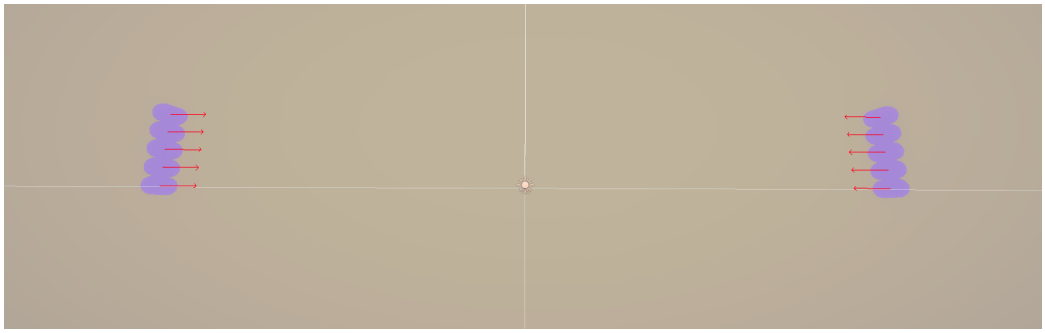


Figure 6.4 Initial layout of the OppositeMultipleScene

6.1.6 OppositeCircleAgents

This scenario is the next iteration of the previous *OppositesMajorScene* scenario and is also a typical test use case in other collision avoidance algorithms. Agents are placed on the arc of the circle with the same spacing between them and destinations that are on the opposite side of the circle. In this scenario, we use 10 agents and a circle with a radius of 10 units. This scenario should test mostly for collisions among the agents, making it exceptionally hard because, in theory, all of their ideal paths have a single intersection point in the centre of the circle. This means that agents need to quickly adapt to others to successfully move to their destination with as few collisions as possible. In our solution, each agent handles its path separately, therefore we expect some collision might occur. The initial layout can be seen in Figure 6.5. Unity's dedicated scene for this scenario in our solution is called *OppositeCircleScene*.

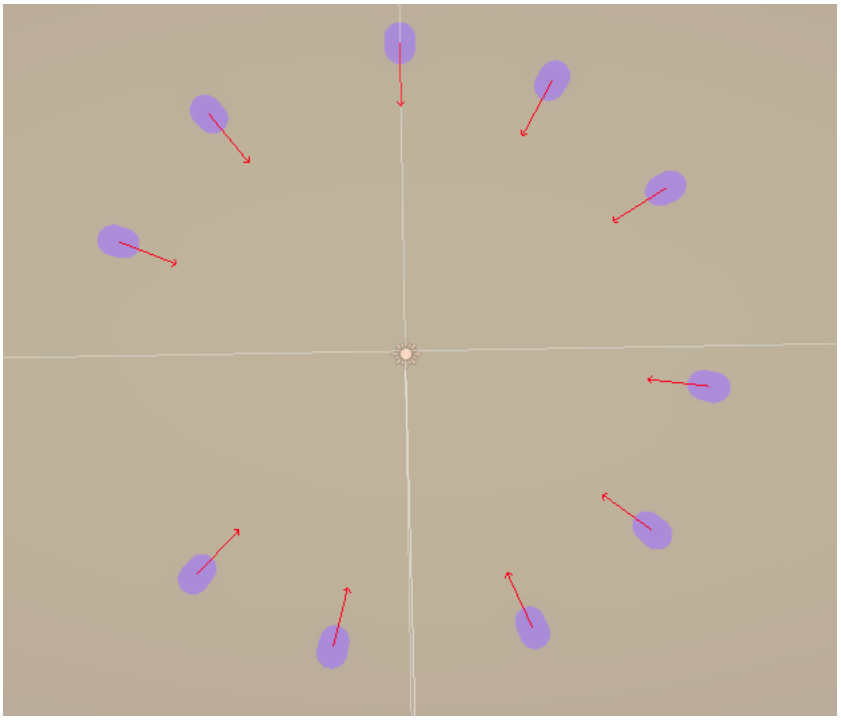


Figure 6.5 Initial layout of the OppositeCircleScene

6.1.7 NarrowCoridorsOppositeNoNavmeshScenario

This scenario is the next iteration of the *SmallObstacleScene* scenario, but it increases difficulty on multiple levels. It consists of 10 agents facing each other similar to the *OppositeMultipleScene* scenario, but the agents have some space between them. Each agent has a destination set to on the opposite side. There are also multiple static obstacles not registered in the navmesh between the agents that form narrow corridors and give agents multiple choices on how to get to the destination. This scenario tests another common problem with collision avoidance algorithms, namely navigating through the same corridor (because it creates the shortest path to the destination), leading to more collisions, rather than selecting a different route. We can see the initial layout in Figure 6.6. Unity's dedicated scene for this scenario in our solution is called *NarrowCorridorNoNavmeshOpposite*.

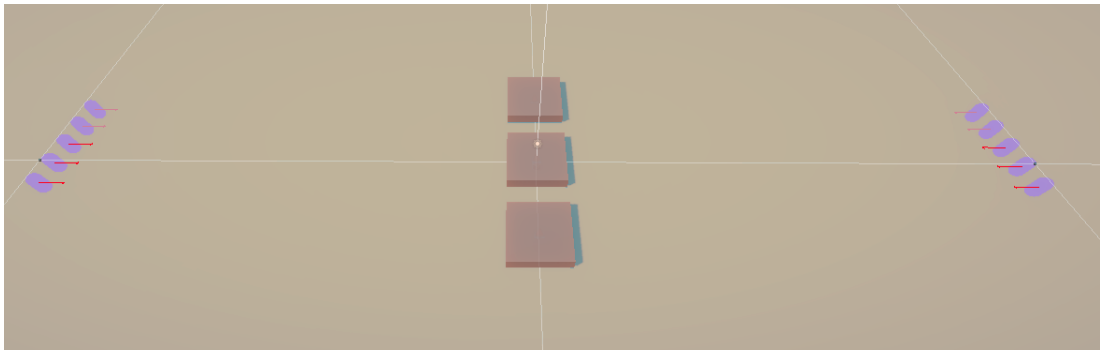


Figure 6.6 Initial layout of the NarrowCorridorNoNavmeshOpposite

6.1.8 NarrowCorridorOpposite

This scenario is a variation of the previous *NarrowCorridorNoNavmeshOpposite* scenario, with a small difference. Now, there is only one narrow corridor and agents are forced to navigate through it (because there are no other paths to the destinations). This gives less space for collision avoidance since agents are no longer able to select a different route. The initial layout is shown in Figure 6.7. Unity's dedicated scene for this scenario in our solution is called *NarrowCorridorOpposite*.

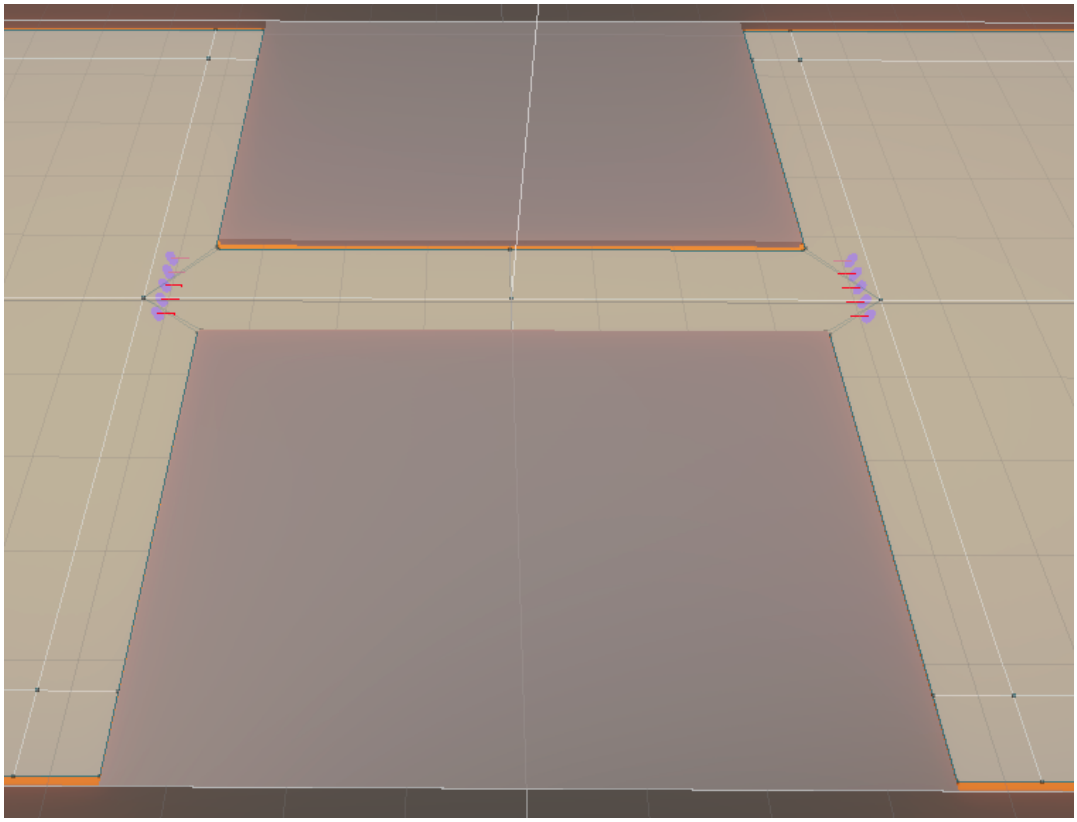


Figure 6.7 Initial layout of the NarrowCorridorOpposite

6.1.9 Hyperparameters

Our solution provides us with multiple configuration options. Changing these options might have an impact on the overall results. We call these options hyperparameters. For this thesis, we selected the following hyperparameters:

1. *ShuffleControlPoints* mutation probability
2. *SmoothAcc* mutation probability
3. *ShuffleAcc* mutation probability
4. *StraightFinish* mutation probability
5. *UniformCross* crossover probability
6. *Collision* fitness weight
7. *EndDistance* fitness weight

8. *JerkCost* fitness weight
9. *TimeToDestination* fitness weight
10. Population size
11. Number of iterations of the GA

Each of these hyperparameters is in a numeric form, and modifying its value might impact the overall performance of our solution. In the following sections, we present experiments that we run to evaluate this impact. We will focus mainly on the metrics of a good path that we defined in the Introduction Section 1.2.

6.2 Results

In this section, we first go through our default hyperparameters configuration, which we refer to as baseline. Then we present the setup of experiments and their results.

6.2.1 Baseline

We refer to the baseline as the default configuration of our solution. It is in the form captured in the table in Figure 6.8

Baseline configuration table	
Hyperparameter	Value
ShuffleControlPoints mutation probability	0.3
SmoothAcc mutation probability	0.9
ShuffleAcc mutation probability	0.3
StraightFinish mutation probability	1
UniformCross crossover probability	0.1
Collision fitness weight	0.5
EndDistance fitness weight	0.2
JerkCost fitness weight	0.2
TimeToDestination fitness weight	0.1
Population size	100
Number of iterations of the GA	20

Figure 6.8 Table describing value of hyperparameters in the baseline configuration.

From our baseline configuration we can observe the following:

- We put high emphasis on collision, since they take 50% of our overall individual fitness. In the second place, we have the path length, because it is the combination of both *EndDistance* and *TimeToDestination* fitnesses (which takes 30% of total individual fitness). The *JerkCost* fitness is last, because thanks to our individual representation as a Bezier curve, we have secured that paths will not be so jittery, therefore its main check is for sudden acceleration/deceleration changes.
- We rely highly on mutations to modify our individual in the best way possible.
- We have a pretty high number of individuals in the population and a bit less iterations.

In the following section, we describe sweeps that we performed across the hyperparameters to determine their impact on the overall solution.

6.2.2 Hyperparameter sweeps

To determine how various hyperparameters affect the overall solution, we perform various sweeps across their values. We then run experiments (for more details, refer to Section 6.2.3) with these modified values to collect data and see what their impact is. We divide sweeps that we performed into two following categories:

1. Non-dependent - these are hyperparameters that can change their value independently on others. This category contains mutation and crossover probabilities, population size, and number of GA iterations.
2. Dependent - these hyperparameters are in dependent relation with others, meaning that if we change one value, we need to also modify some other hyperparameter for our solution to work correctly. This category contains fitness weights, because, as mentioned in Section 5.5.5, sum of their values should be 1.

Sweeps are performed on one hyperparameter at the time, while other hyperparameters have values presented in the Baseline Section 6.2.1. Sweeps for non-dependent hyperparameters are presented in the table in Figure 6.9, where on the left side we have the name of the hyperparameter, and on the right side is an array of values used for sweep (first value in the array is the baseline value).

Sweeps for the non-dependent hyperparameters	
Hyperparameter	Sweep array
ShuffleControlPoints mutation probability	[0.3, 0.6, 0.9]
SmoothAcc mutation probability	[0.9, 0.6, 0.3]
ShuffleAcc mutation probability	[0.3, 0.6, 0.9]
StraightFinish mutation probability	[1, 0.6, 0.3]
UniformCross crossover probability	[0.1, 0.3, 0.6]
Population size	[100, 50, 20]
Number of iterations of the GA	[20, 10, 5]

Figure 6.9 Table describing sweeps across the non-dependent hyperparameters

As mentioned above, dependent sweeps require a change among all dependent values. To capture the configuration of the sweep, we came up with an encoding from which it is easier to tell what kind of sweep it represents. As we know, the dependent hyperparameters are the weights of the fitness functions. The sweep configuration is then encoded as the starting letter of the name of the fitness function, followed by its weight, divided by underscores. The baseline configuration can be then encoded as $C05\_E02\_J02\_T01$, where:

1. $C05$ represents the *Collision* fitness function and its weight 0.5
2. $E02$ represents the *EndDistance* fitness function and its weight 0.2
3. $J02$ represents the *JerkCost* fitness function and its weight 0.2
4. $T01$ represents the *TimeToDestination* fitness function and its weight 0.1

We can see all the configurations of our dependent sweeps with their explanation in the table in Figure 6.10, where on the left side we have the encoded configuration and on the right side explanation of what it encodes.

Sweeps for dependent hyperparameters	
Configuration	Weight values
C05_E03_J01_T01	*Collision* fitness weight 0.5 *EndDistance* fitness weight 0.3 *JerkCost* fitness weight 0.1 *TimeToDestination* fitness weight 0.1
C07_E02_J005_T005	*Collision* fitness weight 0.7 *EndDistance* fitness weight 0.2 *JerkCost* fitness weight 0.05 *TimeToDestination* fitness weight 0.05
C03_E05_J01_T01	*Collision* fitness weight 0.3 *EndDistance* fitness weight 0.5 *JerkCost* fitness weight 0.1 *TimeToDestination* fitness weight 0.1
C02_E07_J005_T005	*Collision* fitness weight 0.2 *EndDistance* fitness weight 0.7 *JerkCost* fitness weight 0.05 *TimeToDestination* fitness weight 0.05
C015_E005_J04_T04	*Collision* fitness weight 0.5 *EndDistance* fitness weight 0.3 *JerkCost* fitness weight 0.1 *TimeToDestination* fitness weight 0.1
C015_E04_J04_T005	*Collision* fitness weight 0.15 *EndDistance* fitness weight 0.4 *JerkCost* fitness weight 0.4 *TimeToDestination* fitness weight 0.05
C02_E005_J005_T07	*Collision* fitness weight 0.2 *EndDistance* fitness weight 0.05 *JerkCost* fitness weight 0.05 *TimeToDestination* fitness weight 0.7
C025_E025_J025_T025	*Collision* fitness weight 0.5 *EndDistance* fitness weight 0.3 *JerkCost* fitness weight 0.1 *TimeToDestination* fitness weight 0.1

Figure 6.10 Table describing sweeps across the dependent hyperparameters

The idea behind each sweep is the following:

1. $C05\_E03\_J01\_T01$ - this configuration is relatively close to the baseline. Half the emphasis is put on the collisions, followed by the path length, and

putting jerk cost at the end. Motivation here is to try to keep the overall balance between collisions and path length while slightly prioritising the collisions.

2. $C07\_E02\_J005\_T005$ - this configuration is putting even more focus on the collisions, but still leaving the path length in the second place and even less focus on the jerk cost. The relationship between collisions and the path length is now more unbalanced. Motivation here is to test how much the given disbalance disrupts the overall behaviour and time it takes the agent to get to the destination.

3. $C03\_E05\_J01\_T01$ - in this configuration, we switch the priority to the path length, putting collisions focus on the second place and still keeping the jerk cost on the third. Motivation is to observe whether there will be a notable increase in the agent's collision count.

4. $C02\_E07\_J005\_T005$ - this configuration is similar in imbalance to the $C07\_E02\_J005\_T005$, but now we switch the collisions with the path length. The jerk cost is still third. Motivation is to find out whether the agent gets faster to the destination while still keeping a reasonable amount of collisions.

5. $C015\_E005\_J04\_T04$ - this configuration switch focusses more on the jerk cost. We can see that the path length still has slightly more importance. This is because we believe that paths cannot be purely focused on jerk cost because that would not necessarily lead agents effectively to their destinations. Motivation is to observe what happens when there is a greater focus on the jerk cost.

6. $C015\_E04\_J04\_T005$ - this configuration is almost the same as $C015\_E005\_J04\_T04$, but we changed the weights of the path length fitnesses. Motivation is the same as in $C015\_E005\_J04\_T04$, but now we are trying it with a different path length fitness.

7. $C02\_E005\_J005\_T07$ - here we are again changing the weights of the path length fitnesses, but now with the $C02\_E07\_J005\_T005$ configuration. Motivation is to test whether a greater focus on the *TimeToDestination* fitness could present better results than focussing on the *EndDestination* fitness.

8. $C025\_E025\_J025\_T025$ - in this configuration we are not prioritising any fitness, setting the same weight to all of them. Motivation is to test an even distribution of focus and the resulting behaviour.

6.2.3 Experiments

In this section, we describe the experiments that were performed to evaluate our solution and how we visualised the results. Each experiment consists of running each configuration on each scenario described through Section 6.1.1 to Section 6.1.8 20 times. The only exception to this is the baseline solution, which we run twice (which is equal to running each scenario 40 times). These runs created logs that are described in Section 6.2.3 and from these logs we created graphs to visualise the data (described in the same section). We use the violin plot for data visualisation (complete guide on how to interpret violin plots is available on Atlassian page (22)). This is mainly because it allows us to see the overall distribution of the data and various peaks. It is also easier to compare these distributions among multiple configurations. Each violin plot is accompanied with box plot for us to more easily see ends of the first and third quartile, as well as median shown by white strip.

The experiments were conducted in a macOS Ventura version 13.4.1 laptop with an Apple M1 chip.

Logging and graph creation

In this section, we first describe an important part of our solution, logging. It is used mainly to obtain the relevant data from the simulation in order to evaluate our solution. We log data per agent in the specific scenario.

The logging is performed in the *csv* file format. An entry in a file can be interpreted as the path properties of an agent. The file has the following columns:

1. PathLength - number of segments in the path.
2. PathDuration - duration in seconds how long it took the agent to go from start to its destination.
3. CollisionCount - number of collisions agent had during moving along the path.
4. FramesInCollision - how many frames the agent spent in collision during moving along the path.
5. PathJerk - JerkCost value of the whole path (for JerkCost calculation refer to the Section 5.5.5).
6. GaTimes - How long (in milliseconds) it took from scheduling the GA to collecting results.

Data are collected after each GA run on each agent separately. To differ among various logs of different configurations of our GA solution, we store them in separate

folders. The names of these folders encode the current configuration of our GA (to understand the following metrics, refer to Section 5.5) and have the following form:

$$CPM - SMM - SHM - CLM - STM - UC - CF - EF - JF - TF - PS - IT$$

where the abbreviations mean the following:

1. *CPM* - probability of *ShuffleControlPoints mutation*
2. *SMM* - probability of *SmoothAcc mutation*
3. *SHM* - probability of *ShuffleAcc mutation*
4. *CLM* - probability of *ClampVelocity mutation*
5. *STM* - probability of *StraightFinish mutation*
6. *UC* - probability of *Uniform crossover*
7. *CF* - weight of the *Collision fitness*
8. *EF* - weight of the *EndDistance fitness*
9. *JF* - weight of the *JerkCost fitness*
10. *TF* - weight of the *TimeToDestination fitness*
11. *PS* - population size
12. *IT* - number of iterations of the GA

To store the logs, we do the following:

1. Create a directory in the form of encoded configuration
2. Create a specific folder inside the configuration directory for each scenario in the form of a scenario name
3. Log each agent in a separate *csv* file in the form *agentId.csv* where *agentId* is the id of a given agent

The resulting path is then of the form $configuration\_encoding/scenario\_name/agent\_id.csv$.

If we run the scenario with the same configurations in the past, we simply append the results to already created *csv* files.

Each configuration directory also contains the configuration file *config.txt*, which consists of the complete breakdown of the configuration encoding with concrete values.

For better visual interpretation of our logged data, we created a `plotting.ipynb` python script (see Attachments C), that transforms the *csv* files into a graph representation. For data manipulation, we use the Pandas library (8). For graph creation, we use both Matplotlib (9) and seaborn (10) python libraries.

In order to have graphs that are easier to read, we changed the names for configurations composed of non-dependent hyperparameters. Instead of dashed-separated encoding (explained earlier in this section), we now use the shorter name of the hyperparameter and its sweep value separated by underscore. Mapping of the shortcuts is the following:

- *ShuffleControlPoints* mutation probability = *cpMut*
- *SmoothAcc* mutation probability = *smoothAccMut*
- *ShuffleAcc* mutation probability = *shuffleMut*
- *StraightFinish* mutation probability = *straightFinish*
- *UniformCross* crossover probability = *controlCross*
- Population size = *popSize*
- Number of iterations of the GA = *iterations*

The whole script is in Jupyter notebook (11) form for more structured code. The script is divided into multiple cells. We now describe the most important ones.

The third cell is used to plot the data for each scenario in each GA configuration. As already mentioned, for graph visualisation, we use the violin plot. The result is a single graph for each column of our logging *csv* file, in every configuration-scenario combination. We can see an example plot for *PathLength* values in *OppositeAgents* scenario (scenario is explained in previous Section 6.1.4) with modified *ShuffleAcc mutation* probabilty in Figure 6.11.

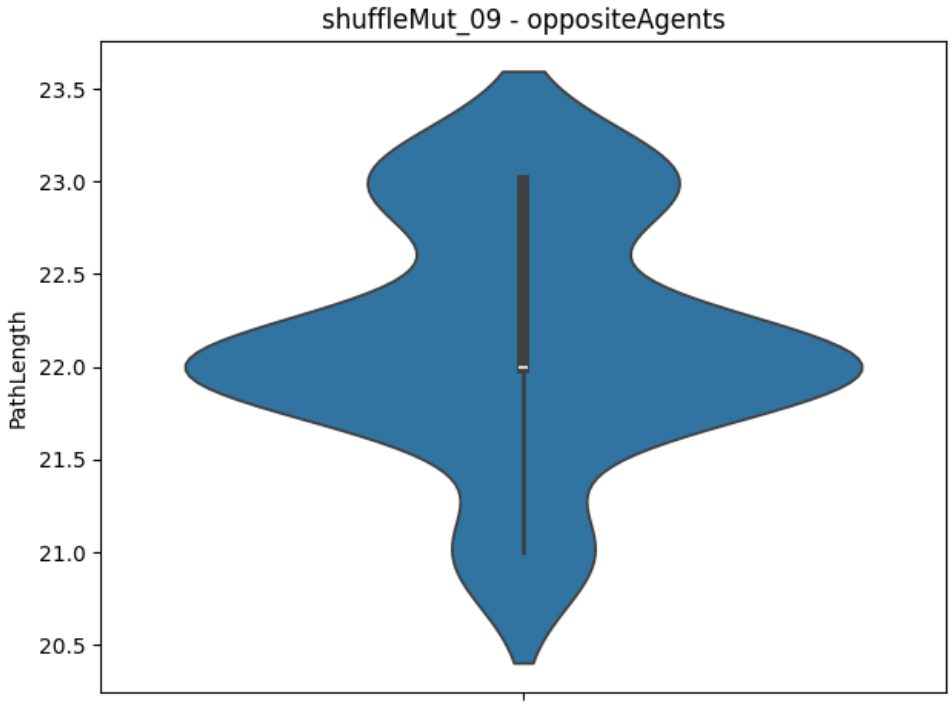


Figure 6.11 Example plot for PathLength values in OppositeAgents scenario with modified ShuffleAcc mutation probability

The fourth cell is used to plot the data for scenario-column combination, where on the x-axis we have different configurations. In this way, we can compare various configurations and their results. The example of this plot is shown in Figure 6.12. It shows various configurations in the *OppositeAgents* scenario (for more details, refer to Section 6.1.4) regarding the *PathLength* values.

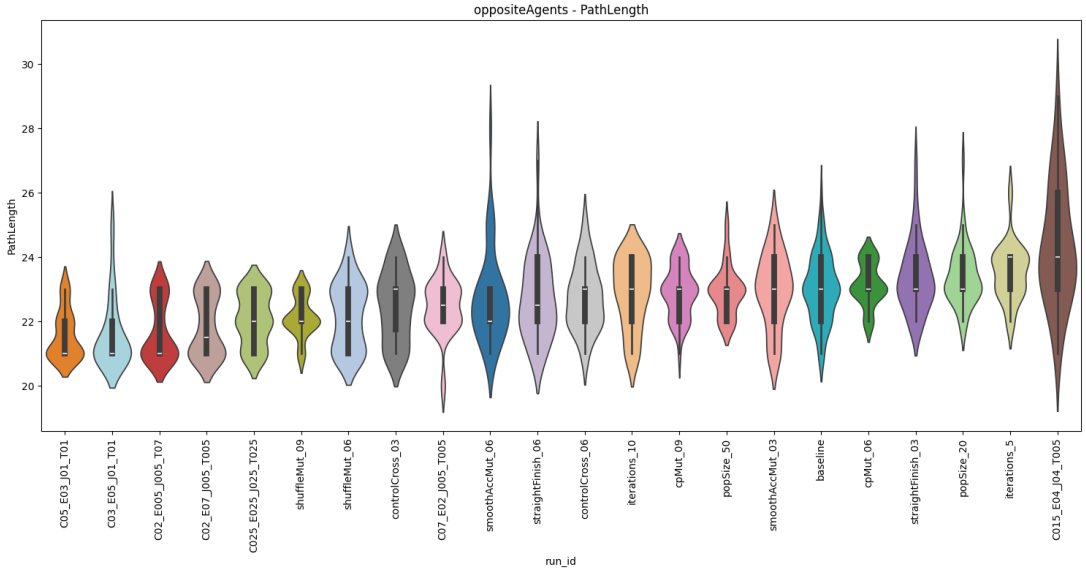
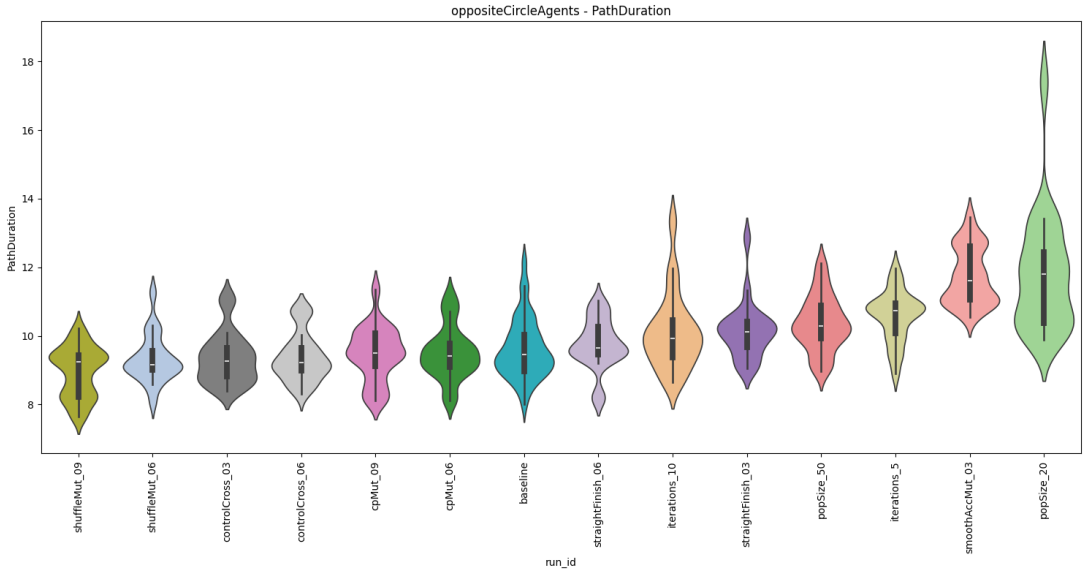
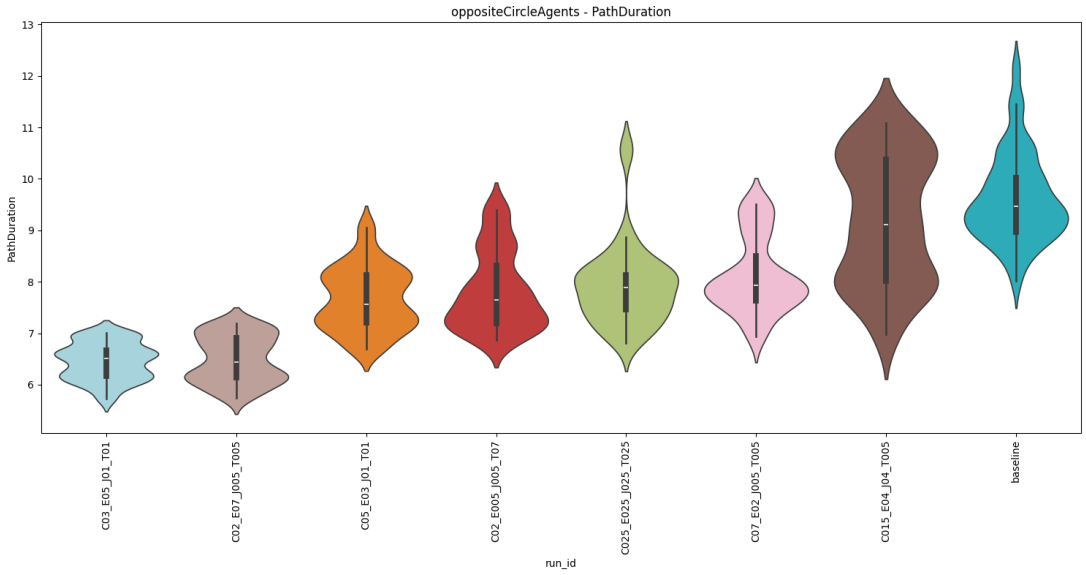


Figure 6.12 Example configurations plot for PathLength values in OppositeAgents scenario

As we can see in Figure 6.12, the configurations plot contains a lot of data. For that reason, we implemented the fifth cell, where we divide configurations into two groups; based on the hyperparameters' dependency. These two groups are then plotted separately, while keeping the same colour encoding (for easier navigation). An example of these graphs can be seen in Figure 6.13.



(a) Configurations composed of the non-dependent hyperparameters



(b) Configurations composed of the dependent hyperparameters

Figure 6.13 Configurations comparison of the PathDuration entry in the OppositeCircleAgents divided into two groups based on hyperparameters' dependency

How to read graphs

As described in Section 6.2.3, our logs store the following data (to which we refer as *graph data*):

1. PathLength
2. PathDuration
3. CollisionCount
4. FramesInCollision
5. PathJerk
6. GaTimes

Our plotting script (described in Section 6.2.3) then visualises each entry from the *graph data* for each scenario separately. It is capable of creating two types of graphs:

1. Configuration-centric graphs - this is the visualisation just for one configuration. It creates separate graphs for each entry in the *graph data* in each scenario. The title of each graph contains the configuration name and the

scenario name separated by a dashed line. An example of such a graph visualising the *PathLength* entry for the *shuffleMut_09* configuration in the *OppositeAgents* scenario is presented in Figure 6.14.

2. Configuration compare graphs - It creates separate graphs for each entry in the *graph data* in each scenario, but now the graph contains all the configurations (the configuration name is in comparison to the baseline solution). The violin plots for these configurations are sorted in ascending order based on their mean value (meaning that configurations more to the left are better). The title of each graph contains the name of the scenario and a concrete entry of the *graph data* separated by a dashed line. An example of a graph visualising the *PathLength* entry in the *OppositeAgents* scenario can be seen in Figure 6.15. You may notice that the $C015\_E005\_J04\_T04$ configuration is missing in this graph (and in any subsequent graphs as well). The reason is explained in Section 6.3.

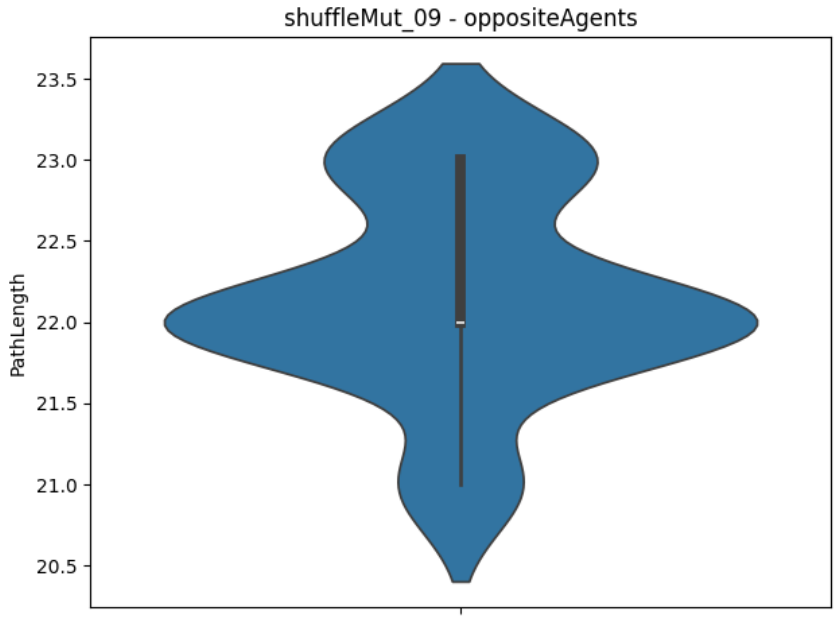


Figure 6.14 PathLength entry for shuffleMut_09 in the OppositeAgents scenario

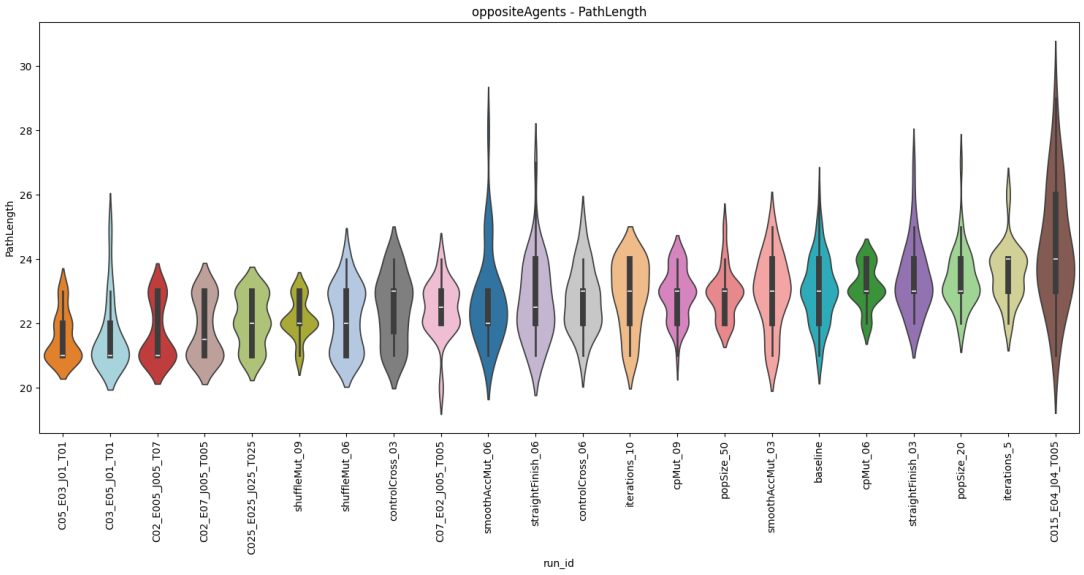


Figure 6.15 Configurations comparison of the PathLength entry in the OppositeAgents scenario

In the following sections, we show several plots. We focus mainly on configuration comparison graphs since they provide us with more information about the differences between them. Showing the plot with every configuration present might not be ideal; therefore, we present the plots in groups of three. The first plot will consist of every configuration to get an overall idea of the order. The next plot will consist only of configurations composed of dependent hyperparameters along with the baseline solution, and the third plot will have baseline solution with comparison of other configurations composed of non-dependent hyperparameters. Note that if the order of some configurations is not consistent across plots, it means that they have the same mean value, in which case we do not have control over ordering.

As we do not want to overwhelm this thesis and its reader with many plots, we show only a small subset of generated graphs. The rest of the plots can be found in the Attachments C to this thesis. For more readability, we divide the plots into the following three categories:

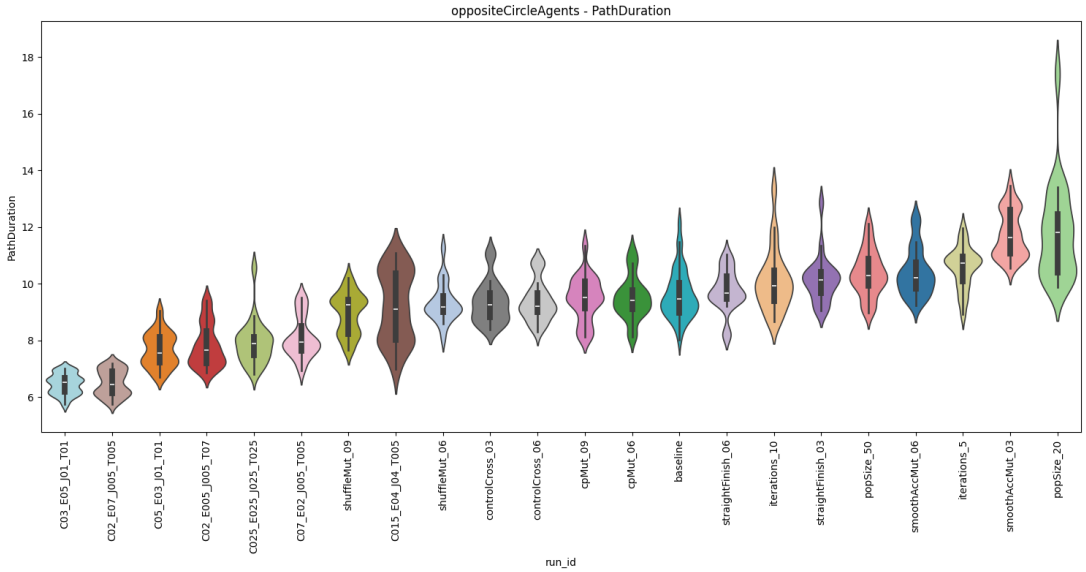
1. Path length - plots related to the path length. These are reflected in *PathLength* and *PathDuration* entries from our *graph data*.

2. Collisions - plots related to agents' collisions. These are reflected in *CollisionCount* and *FramesInCollision* entries from our *graph data*.

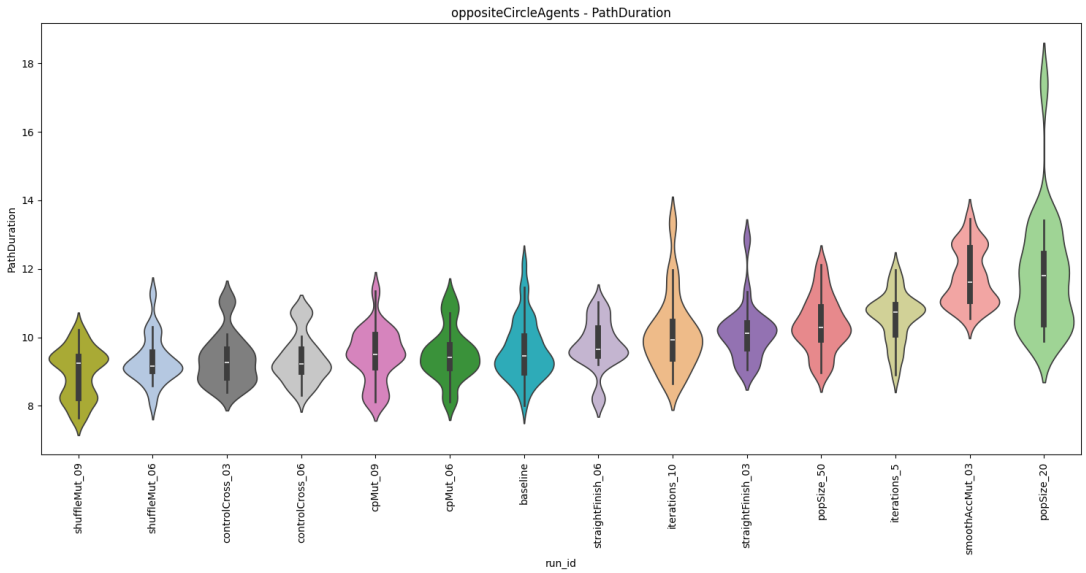
3. Other metrics - plots related to remaining metrics. These are reflected in *PathJerk* and *GaTimes* entries from our *graph data*.

Path length

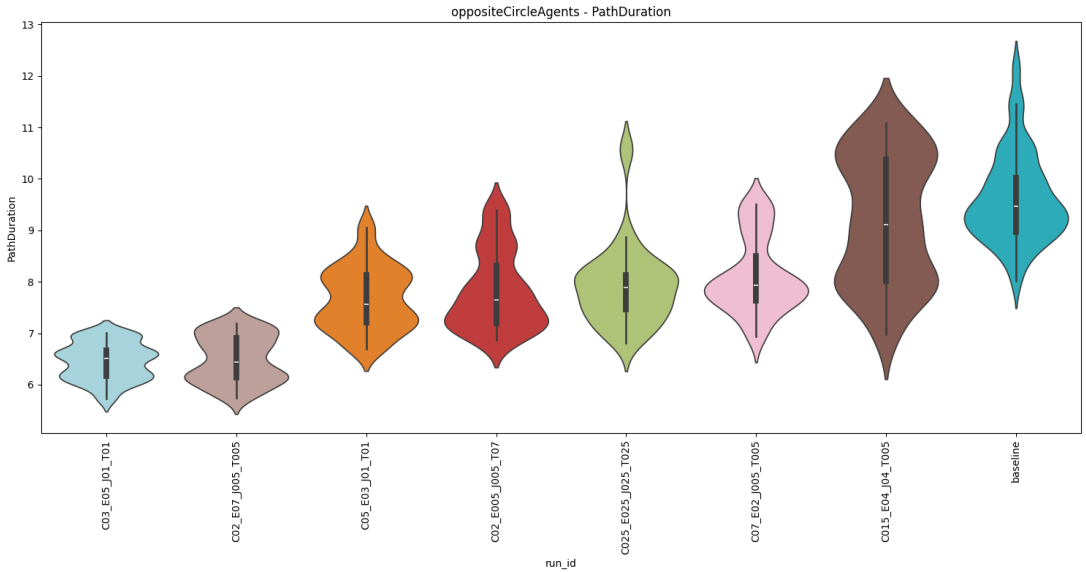
We start with a graph visualising the *PathDuration* entry in the *OppositeCircleAgents* scenario. This can be seen in Figure 6.16



(a) Comparison of all the configurations



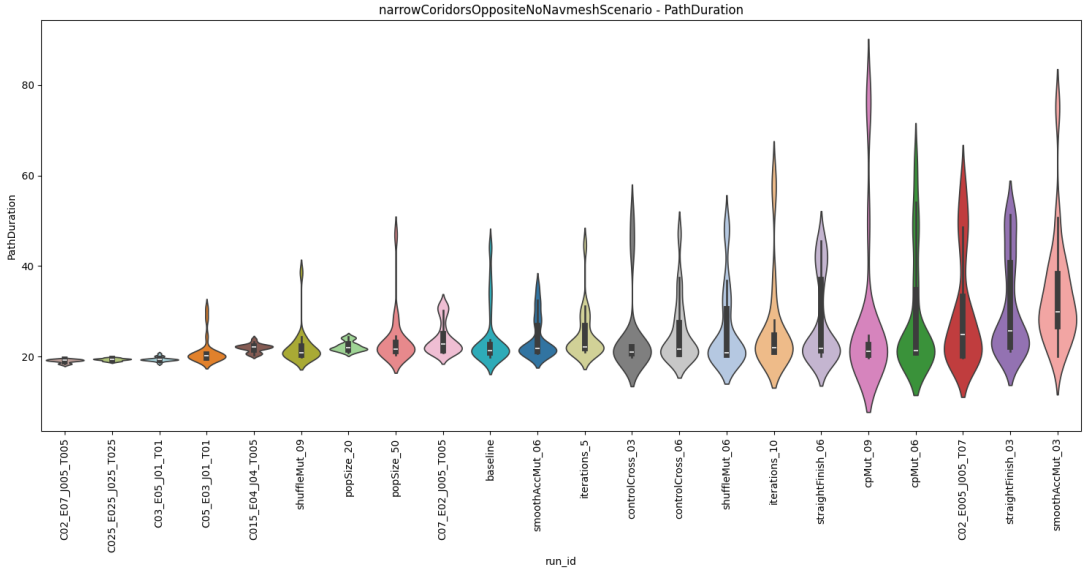
(b) Configurations composed of the non-dependent hyperparameters



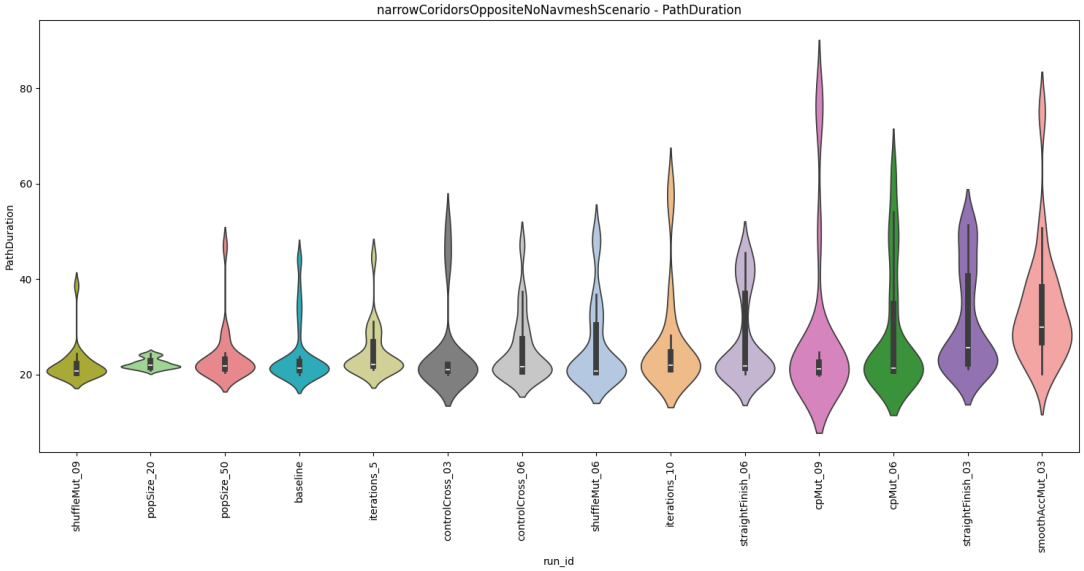
(c) Configurations composed of the dependent hyperparameters

Figure 6.16 Configurations comparison of the PathDuration entry in the OppositeCircleAgents

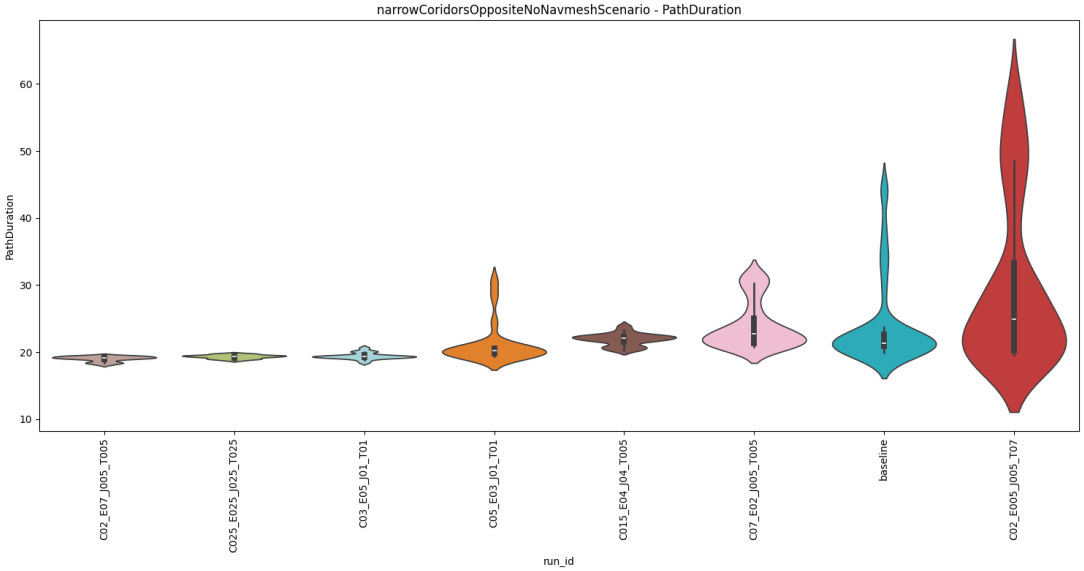
Next, we can see plots for the *PathDuration* entry in the *NarrowCoridorsOppositeNoNavmeshScenario* in Figure 6.17.



(a) Comparison of all the configurations



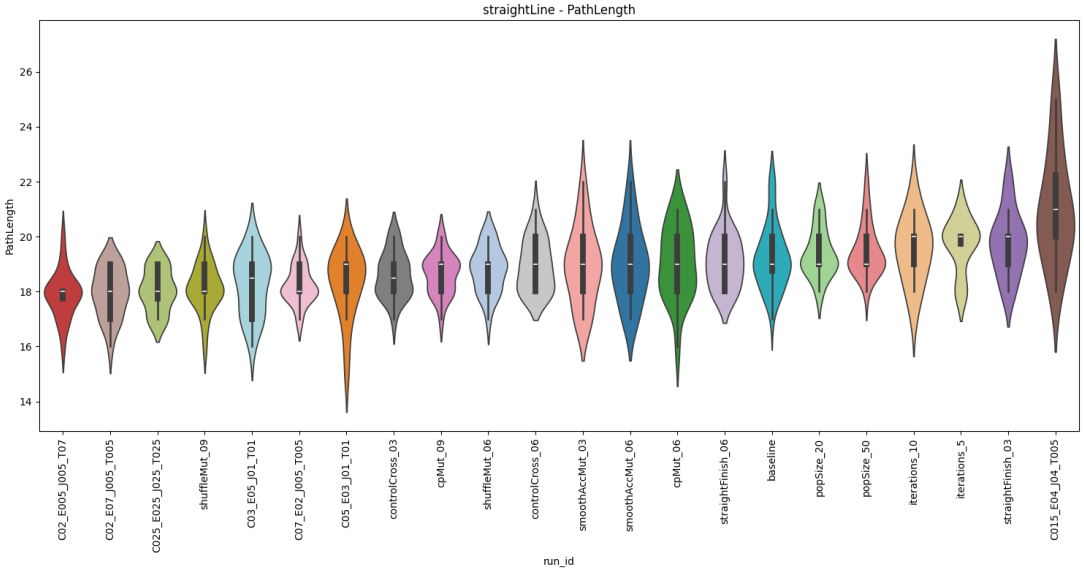
(b) Configurations composed of the non-dependent hyperparameters



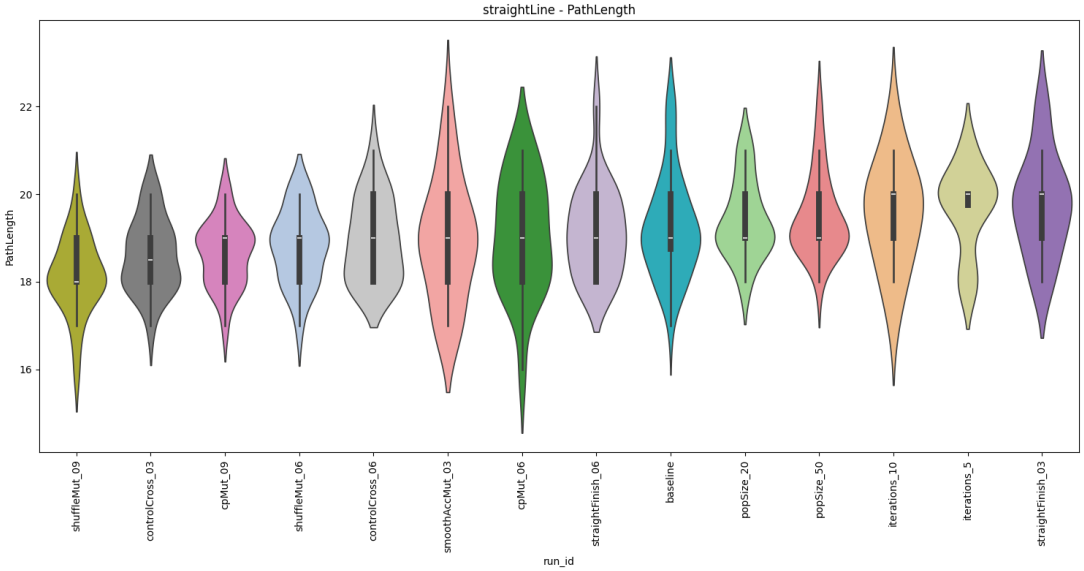
(c) Configurations composed of the dependent hyperparameters

Figure 6.17 Configurations comparison of the PathDuration entry in the NarrowCoridorsOppositeNoNavmeshScenario

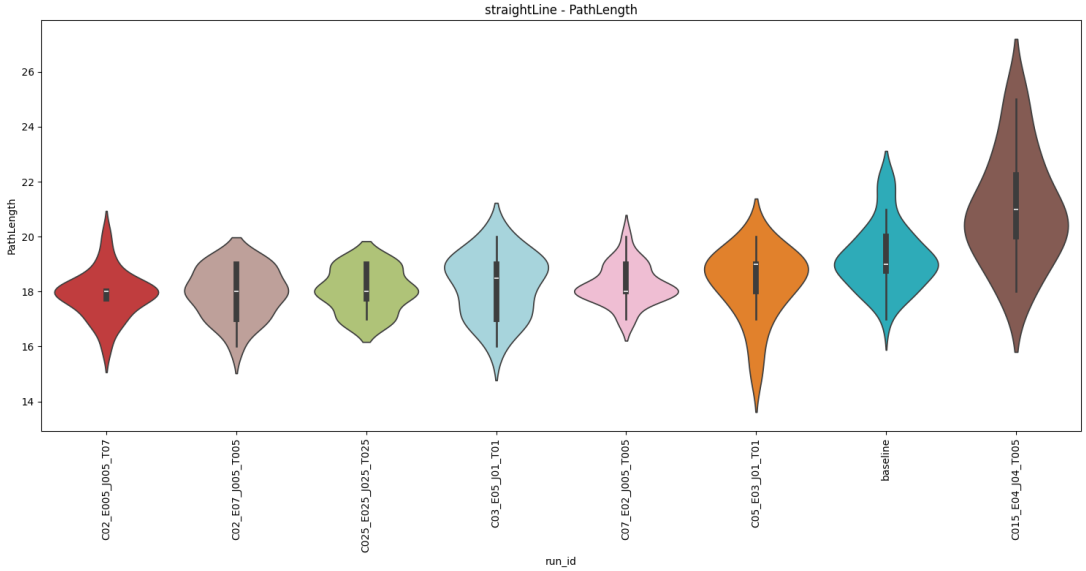
Last of the Path length section is the *PathLength* entry in the *StraightLine* scenario, which can be seen in Figure 6.18.



(a) Comparison of all the configurations



(b) Configurations composed of the non-dependent hyperparameters

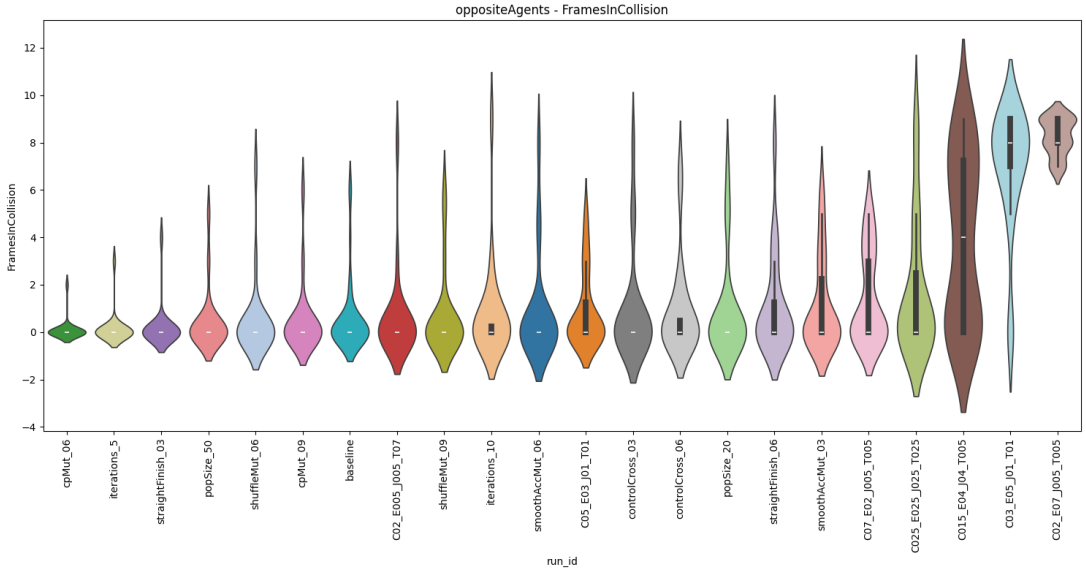


(c) Configurations composed of the dependent hyperparameters

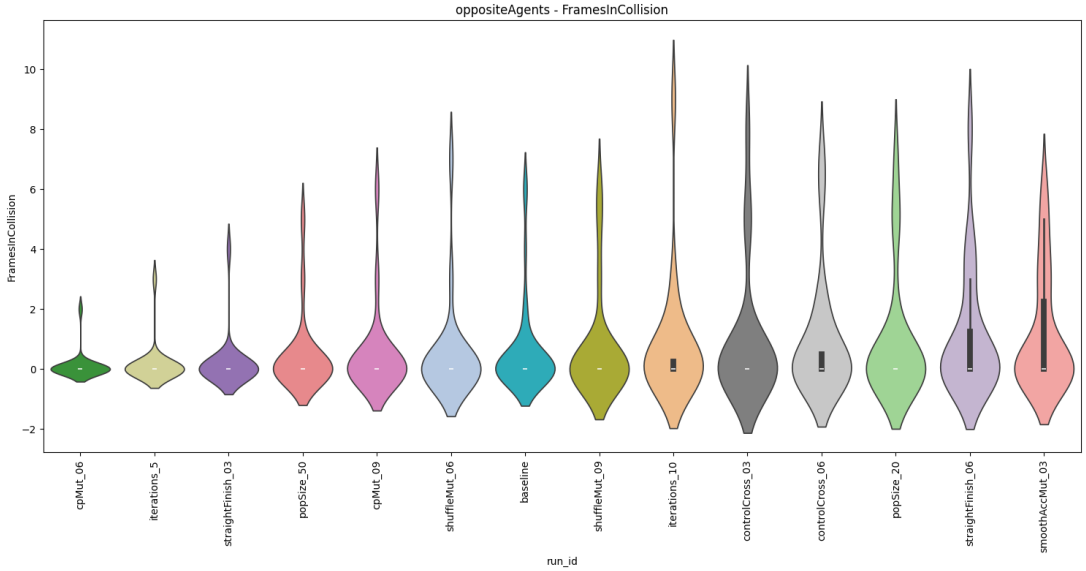
Figure 6.18 Configurations comparison of the PathLength entry in the StraightLine

Collisions

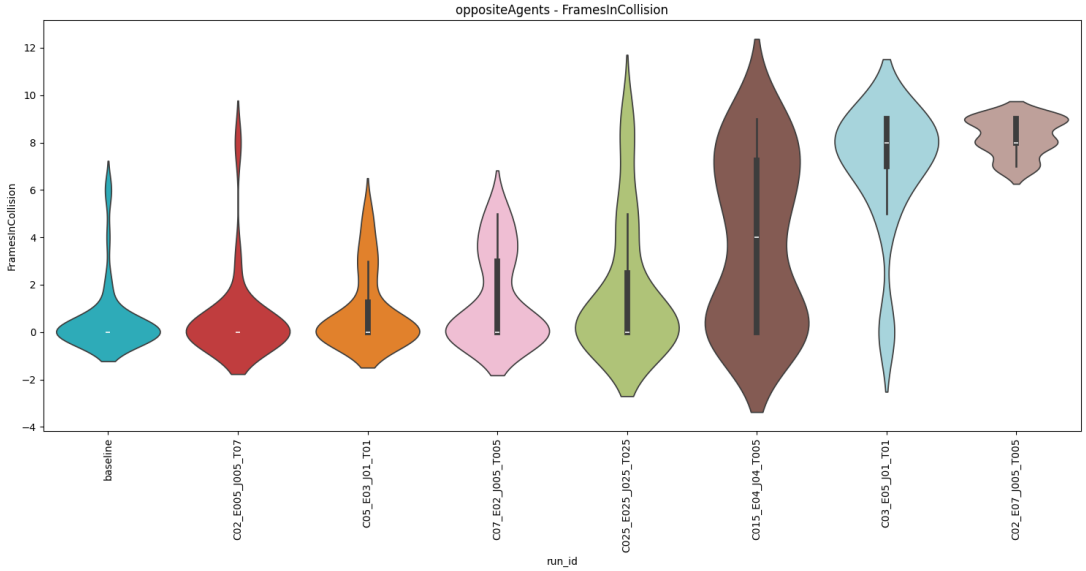
This section starts with the visualisation of the *FramesInCollision* entry in the *OppositeAgents* scenario. It is present in Figure 6.19.



(a) Comparison of all the configurations



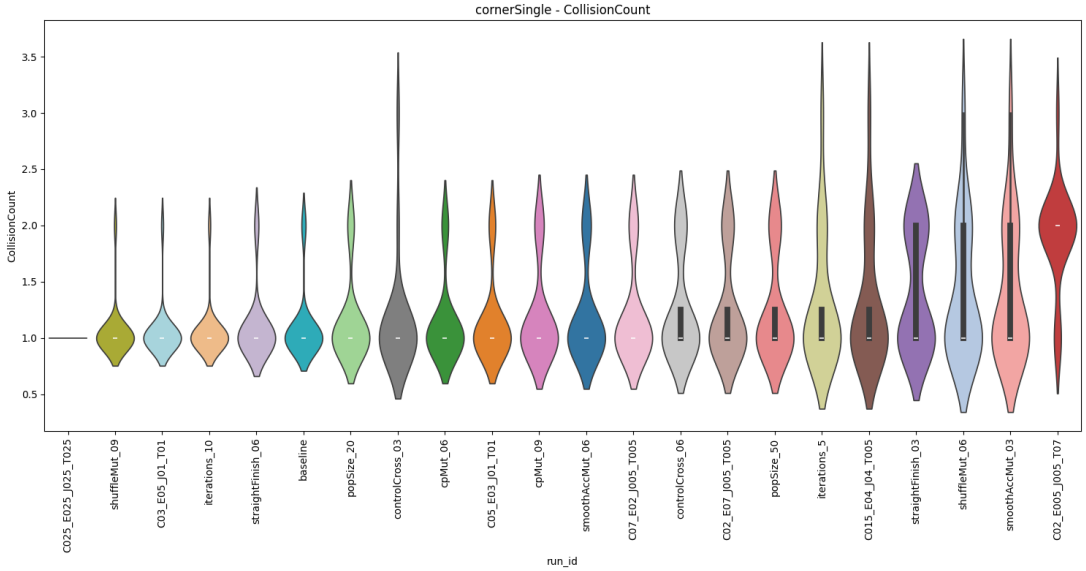
(b) Configurations composed of the non-dependent hyperparameters



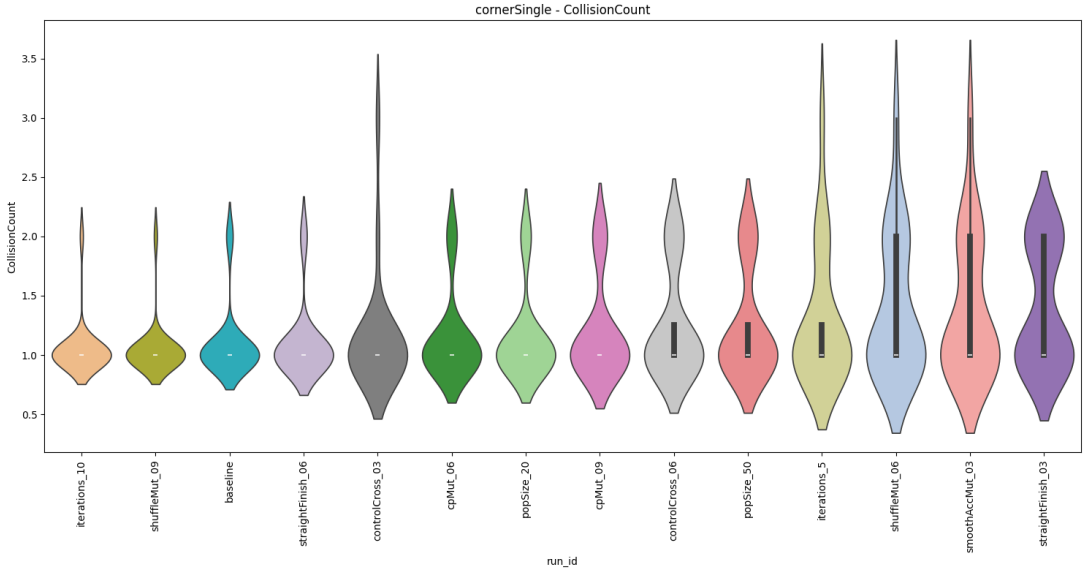
(c) Configurations composed of the dependent hyperparameters

Figure 6.19 Configurations comparison of the FramesInCollision entry in the OppositeAgents

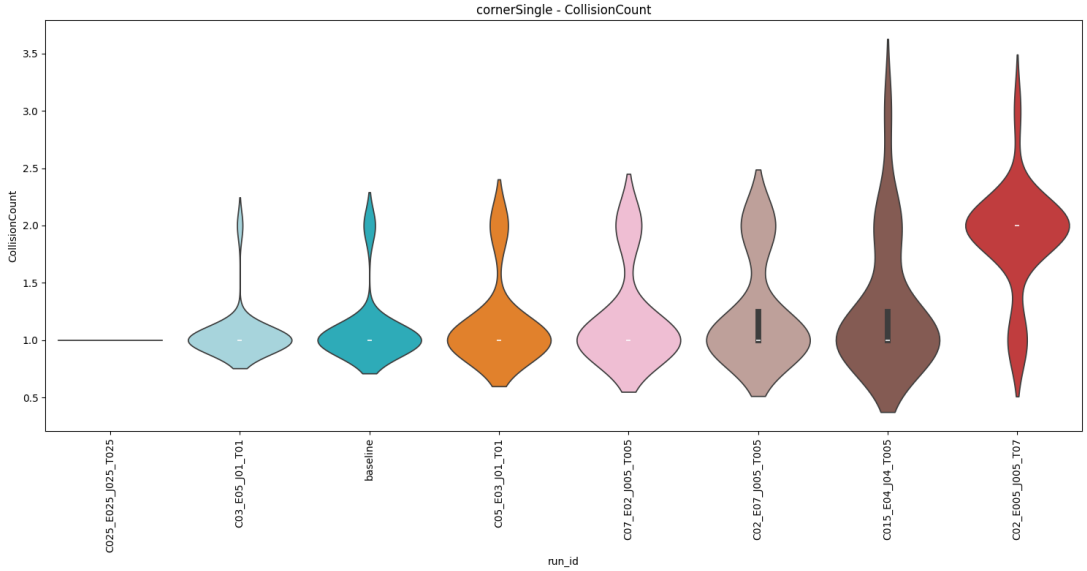
Second visualisation consists of *CollisionCount* entry in *CornerSingle* scenario and can be seen in Figure 6.20



(a) Comparison of all the configurations



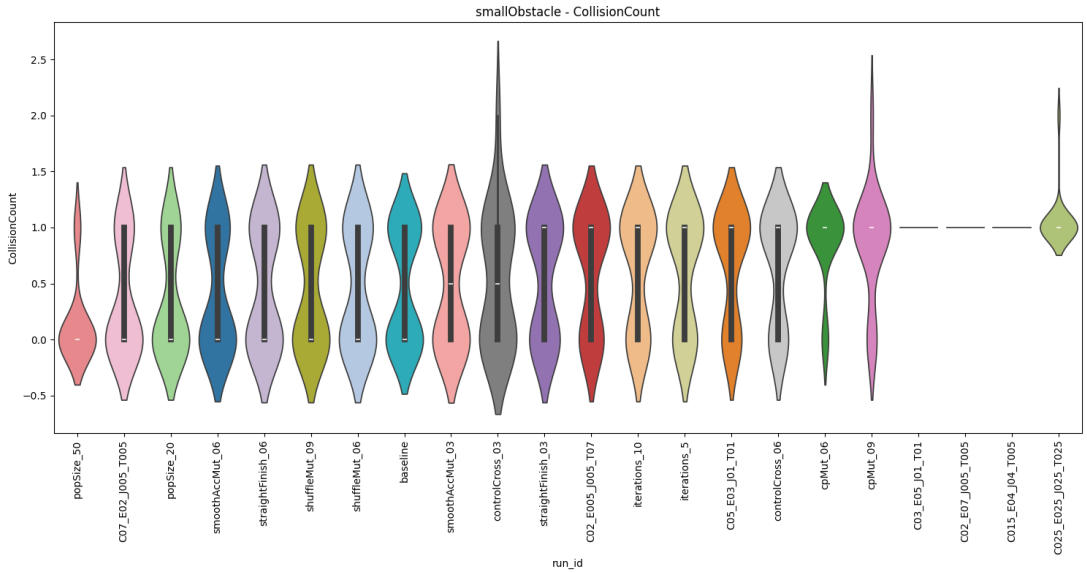
(b) Configurations composed of the non-dependent hyperparameters



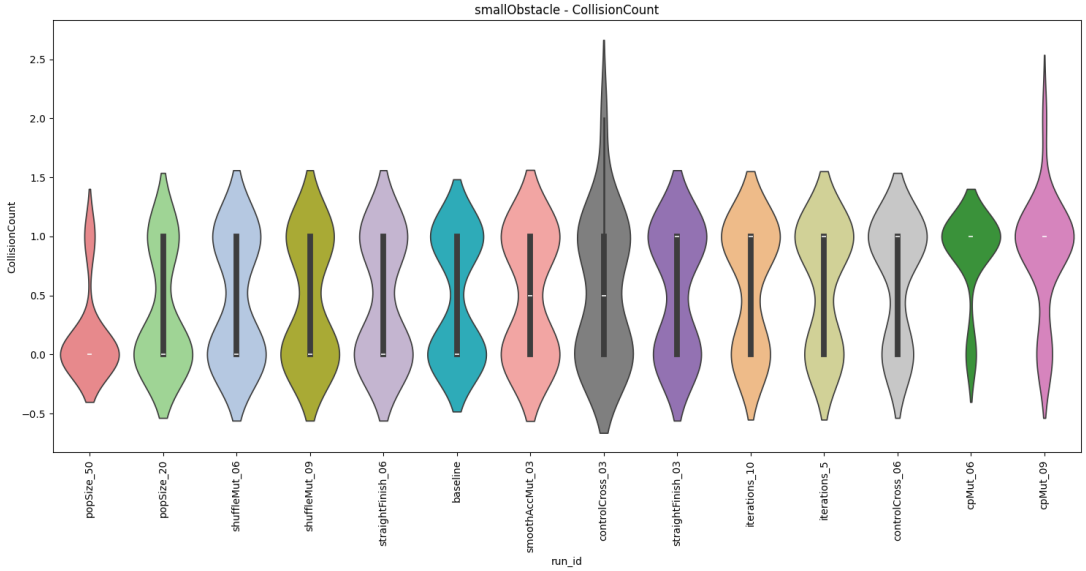
(c) Configurations composed of the dependent hyperparameters

Figure 6.20 Configurations comparison of the CollisionCount entry in the CornerSingle

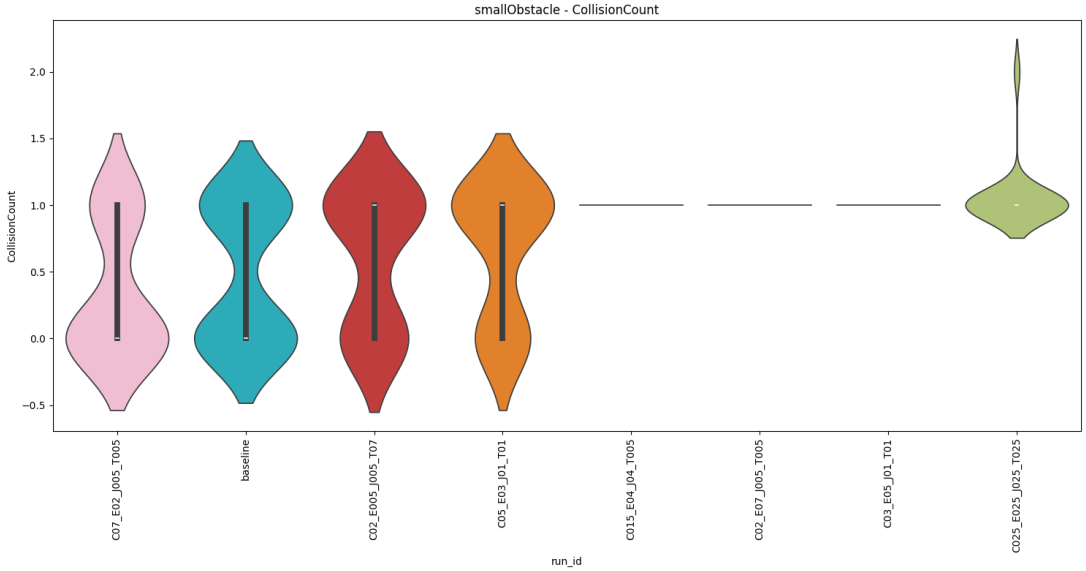
Next, we continue with the *CollisionCount* entry but in the *SmallObstacle*. The results are shown in Figure 6.21



(a) Comparison of all the configurations



(b) Configurations composed of the non-dependent hyperparameters

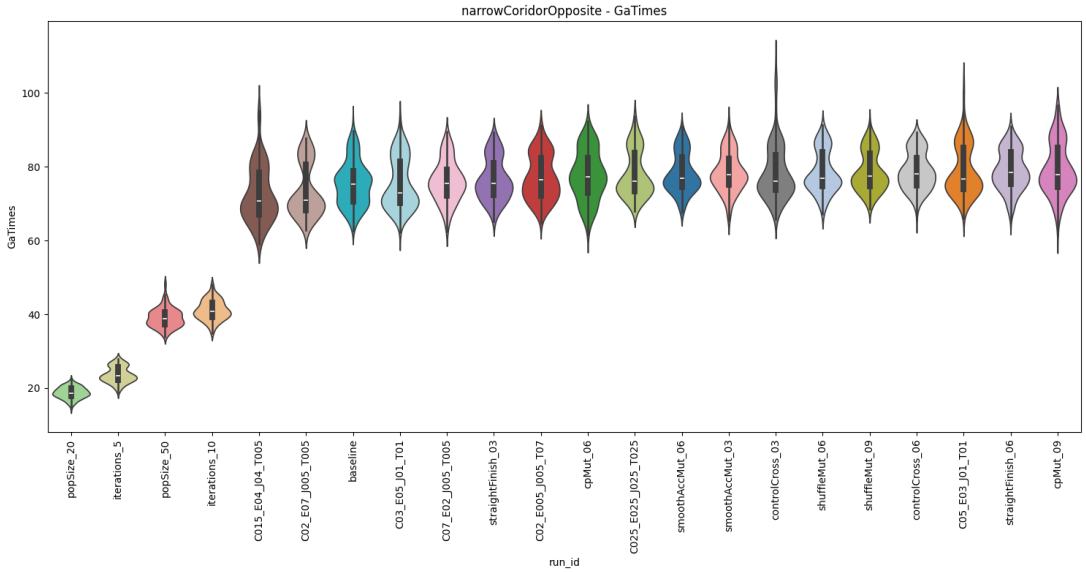


(c) Configurations composed of the dependent hyperparameters

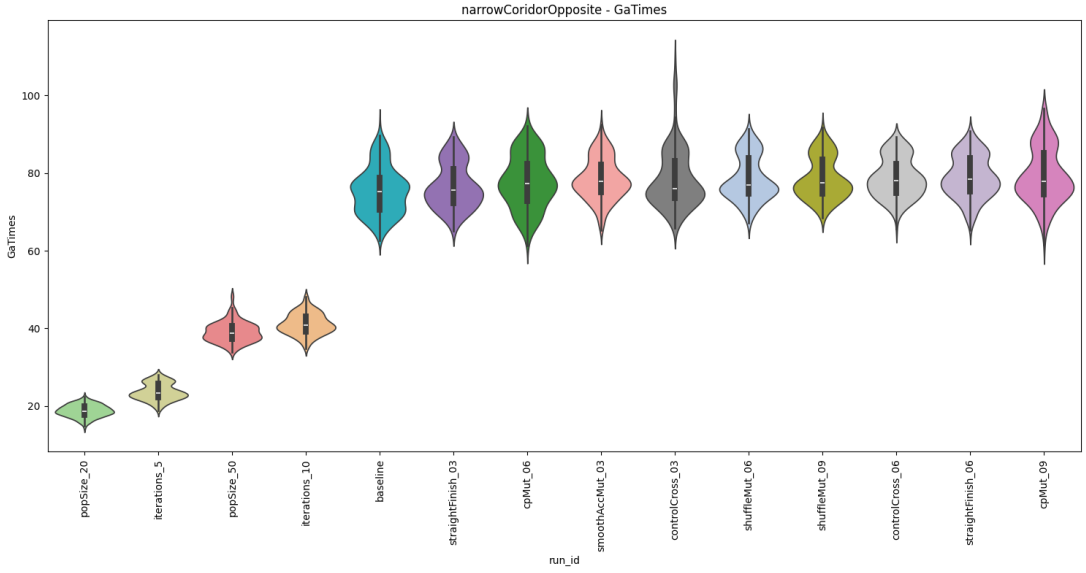
Figure 6.21 Configurations comparison of the CollisionCount entry in the SmallObstacle

Other metrics

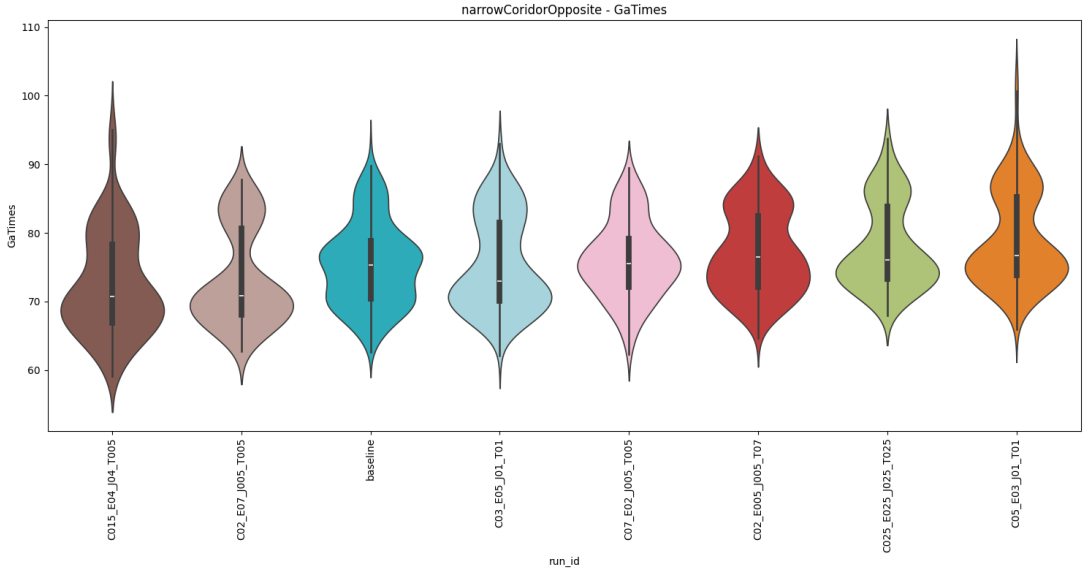
In this section we first show in Figure 6.23 visualisation of the *GaTimes* entry in *NarrowCoridorOpposite*.



(a) Comparison of all the configurations



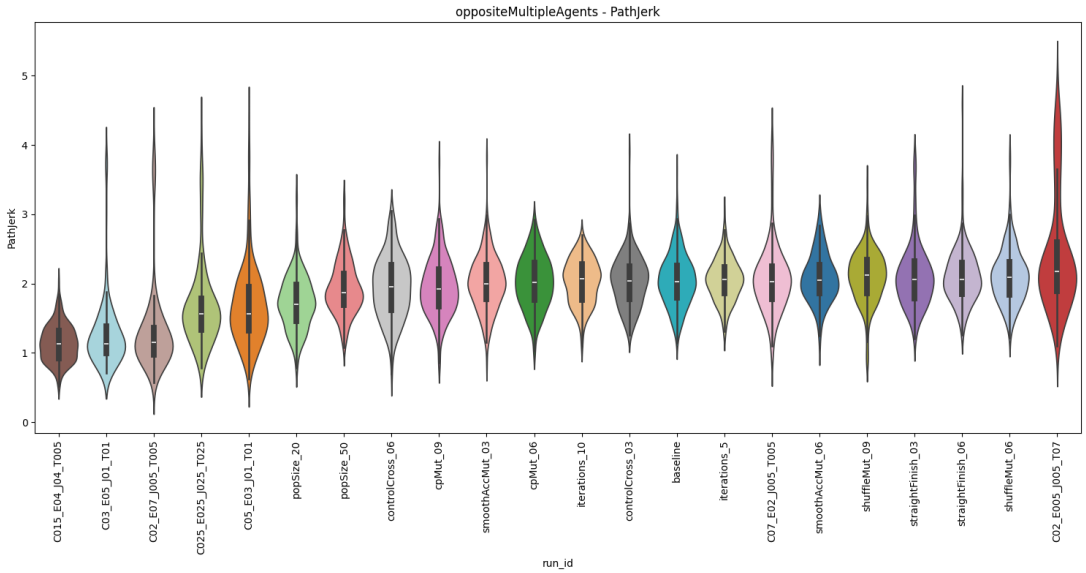
(b) Configurations composed of the non-dependent hyperparameters



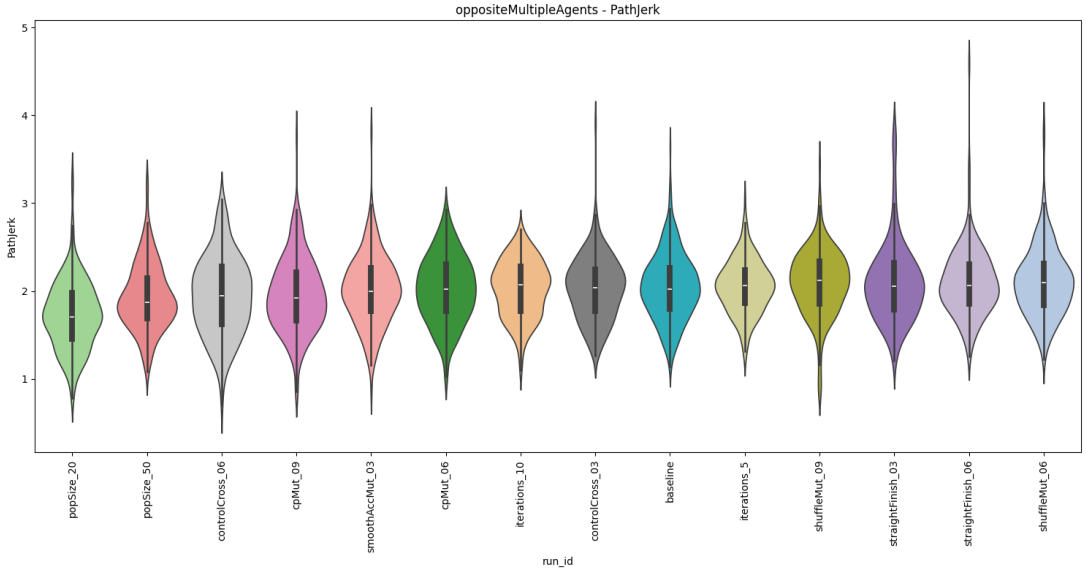
(c) Configurations composed of the dependent hyperparameters

Figure 6.22 Configurations comparison of the GaTimes entry in the NarrowCorridorOpposite

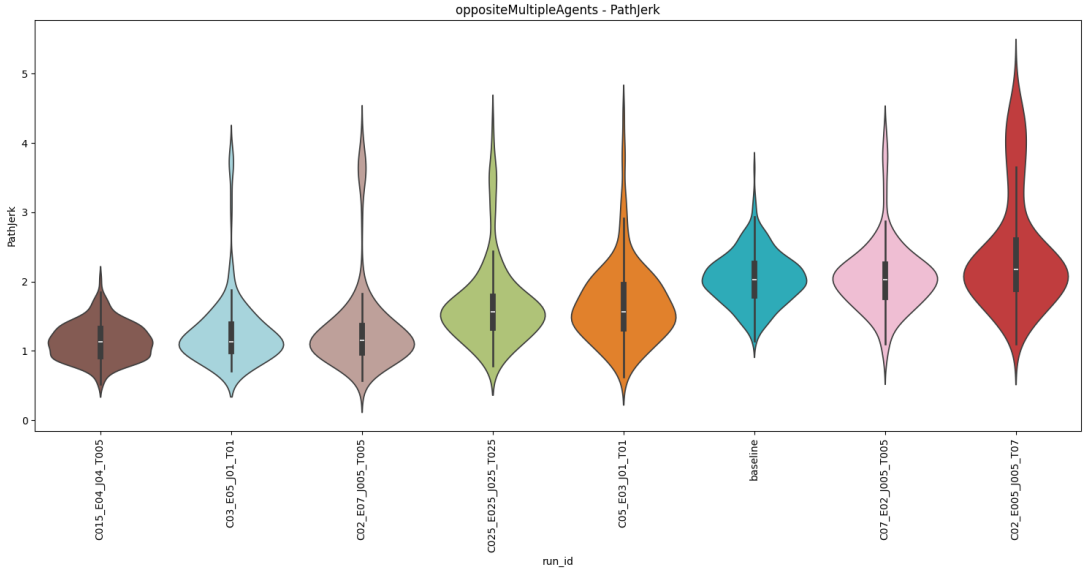
Lastly, we show visualisation of *PathJerk* entry in *OppositeMultipleAgents* in Figure



(a) Comparison of all the configurations



(b) Configurations composed of the non-dependent hyperparameters



(c) Configurations composed of the dependent hyperparameters

Figure 6.23 Configurations comparison of the PathJerk entry in the OppositeMultipleAgents

6.3 Discussion

In this section, we go one visualisation after, look at the results and interpret it the way we see it.

Before that, we need to explain why the configuration *C015_E005_J04_T04* is missing in the resulting plots. This is because agents were unable to complete some scenarios in a reasonable time. The example scenario is *StraightLine*. Since the agent in this scenario is at the beginning 40 units away from its destination, he put the most emphasis on the *JerkCost* fitness value. This resulted in the agent slowly wandering around its start position since it did not want to disturb the *JerkCost* value by accelerating. This is why we decided to omit this configuration from the overall experiments.

6.3.1 OppositeCircleAgents - PathDuration

We can see in Figure 6.16a that the performance of the configurations is relatively similar to each other.

Configurations that perform worse compared to the baseline are generally those that lower the quality of the solution. We can see that these are all configurations that reduce the population size or the number of iterations of the GA. This can have some impact due to the algorithm not having enough viable individuals or not being able to iterate to better solution before it ends. We can also see that *smoothAccMut_03* performed worse than the baseline. This mutation is expected to balance accelerations/decelerations of the agent, keeping him on a similar speed during his path navigation. In this configuration, mutation is executed a lot less than usual (0.9 probability), which might result in irregular accelerations/decelerations and therefore even higher path duration time.

On the other hand, configurations *C02_E07_J005_T005* and *C03_E05_J01_T01* perform better with a noticeable difference. Something these configurations have in common is a relatively low *Collision* fitness weight. It might be worth taking a look at the *FramesInCollision* entry in this scenario to see if these configurations do not prefer the better path duration but neglect collisions. We can see the configurations comparison for this entry in Figure 6.24. Although the configurations seem to have a similar distribution, we can see that configurations *C02_E07_J005_T005* and *C03_E05_J01_T01* are among the worst performers. One more observation that can be done, is that the *C02_E005_J005_T07* configuration is among the best performers in both *FramesInCollision* and *PathDuration* entries, even though it puts higher emphasis on *TimeToDestination* fitness and a lot less on *Collision* fitness. This might be due to the fact that circle on which agents are positioned is relatively small in diameter - 20 units. Therefore, agents might be able to construct paths that end directly at destination in the early beginnings of

their navigation. These paths can then have the same *TimeToDestination* fitness value, therefore preferring the ones with fewer collisions.

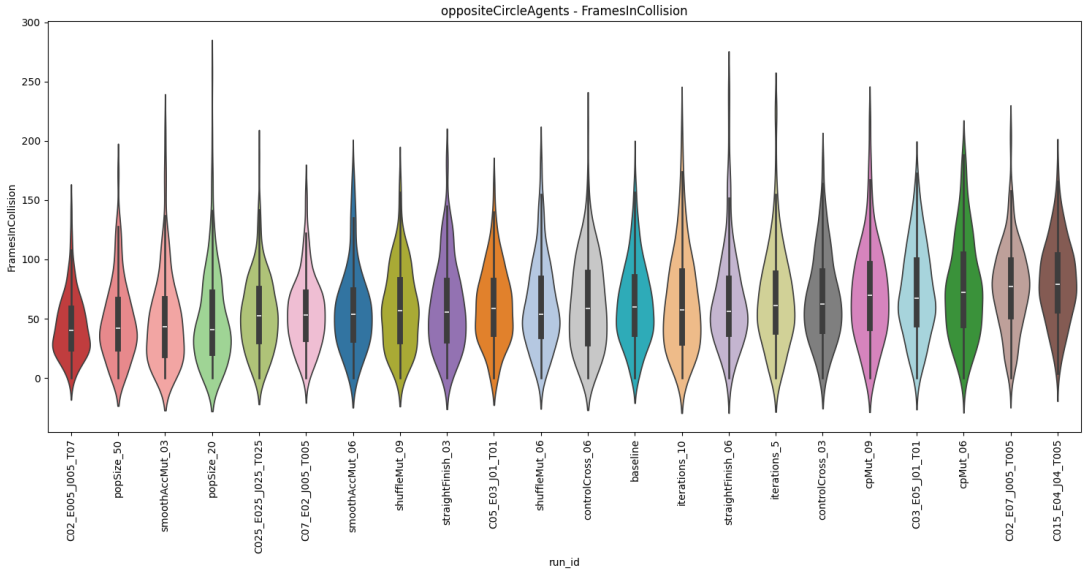


Figure 6.24 Configurations comparison of FramesInCollision entry in OppositeCircleAgents

6.3.2 NarrowCoridorsOppositeNoNavmeshScenario - PathDuration

We can again see in Figure 6.17a that the configurations perform similarly. Another observation we might have is that in almost every configuration there is a tiny percentage of runs that produced bad results. This indicates that some agents are stuck for a while and only then reach the destination. When the experiments were run, we were able to observe that behaviour in some cases in this concrete scenario. Unfortunately, it is due to the nature of our solution. Sometimes an agent gets in a position where he is really close to some wider obstacle and is heading towards that obstacle. If in this case the destination is on the other side of a given obstacle, all paths that the agent produces result in an almost immediate collision. Therefore, the agent might prefer small steps that do not cause collisions, which results in him moving slowly around the obstacle. The agent eventually reaches the point where he is able to find a path without any collisions and continues its navigation more efficiently.

We can see in Figure 6.17a that there are again some leading configurations that do not exhibit this behaviour. We are talking mainly about *C02_E07_J005_T005*, *C025_E025_J025_T025*, and *C03_E05_J01_T01*. We

can see that they again have less emphasis on collisions, but put relatively high focus on path length with a combination of the entries *EndDistance* and the *TimeToDestination*, which might help them overcome this stuck issue.

6.3.3 StraightLine - PathLength

As we can see in Figure 6.18a all configurations perform relatively the same. There is a small exception if we look at the *C015_E04_J04_T005* configuration. We can see that it has some noticeable percentage of runs that performed worse than average. This is most likely due to similar reason why the configuration *C015_E005_J04_T04* was skipped (for more details, see beginning of Section 6.3). The *C015_E04_J04_T005* configuration is mainly focused on the path length, but it has almost the same weight as the *JerkCost* fitness. Since in the *StraightLine* scenario the agent is at the beginning 40 units away from the destination, the *TimeToDestination* fitness values will be the same, leaving the *EndDestination* and the *JerkCost* fitness values equally important. This might result in the agent having slower starts, which then leads to longer paths.

On the other hand, if we look at the top 3 leading configurations, which are *C02_E005_J005_T07*, *C02_E07_J005_T005*, and *C025_E025_J025_T025* respectively, we can see that they all put relatively small emphasis on the *Collision* fitness, which is not really important in this scenario since there are no other agents or obstacles, therefore leaving extra space to more appropriate fitnesses.

6.3.4 OppositeAgents - FramesInCollision

As we can see in Figure 6.19a, almost all configurations have a mean value of *FramesInCollision* equal to zero. We can also observe small peaks among most of the configurations, but these are at most of value 12. Twelve frames in collision is a really small amount of time, so we can almost certainly say that these were not head-on collisions. It was most likely a tight pass that resulted in a collision.

If we look at the last three configurations, which are *C02_E07_J005_T005*, *C03_E05_J01_T01* and *C015_E04_J04_T005*, we can see that they all put small attention to *Collision* fitness. Therefore, it is not surprising that they are not achieving the best results in this scenario.

6.3.5 CornerSingle - CollisionCount

As presented in Figure 6.20a, we can see that most of our configurations have a mean value of the collision count of at least one. Contra-inductively, if we look at how the worst configuration (*C02_E005_J005_T07*) performed in the *FramesInCollision* entry (Figure 6.25), we can see that it has the best results. It

has approximately 50 frames in collision as a mean value. If we divide this value by two (because of the two collisions), we get about 25 frames per collision. This is probably just a small corner cut for each collision. Furthermore, if we look at Figure 6.26, we can see that configuration *C02_E005_J005_T07* is also the best in the *PathDuration* entry. So, even though it has the most collisions, it might not be the worst configuration for this given scenario.

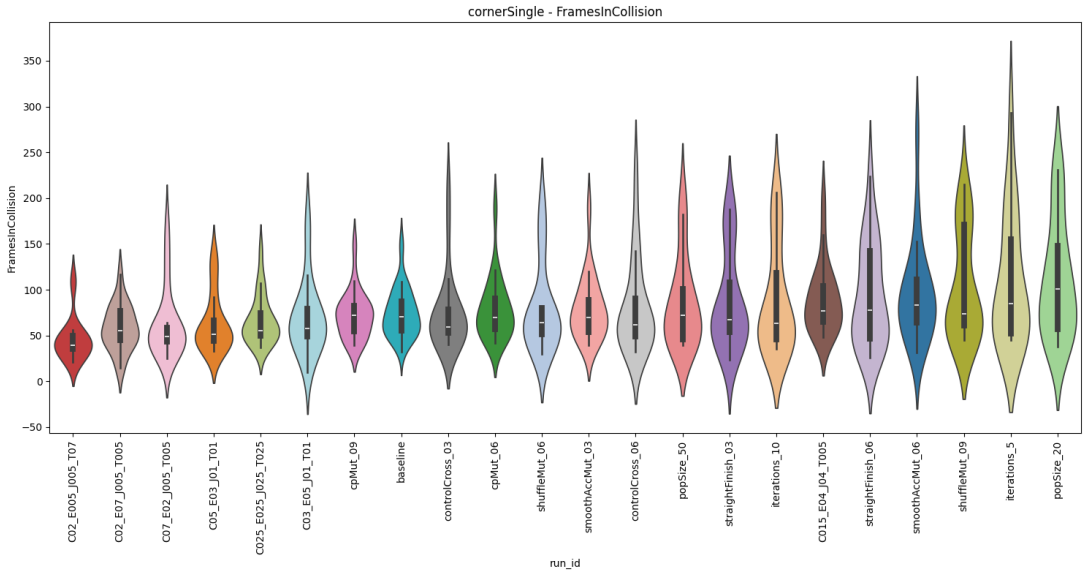


Figure 6.25 Configurations comparison of FramesInCollision entry in CorneSingle

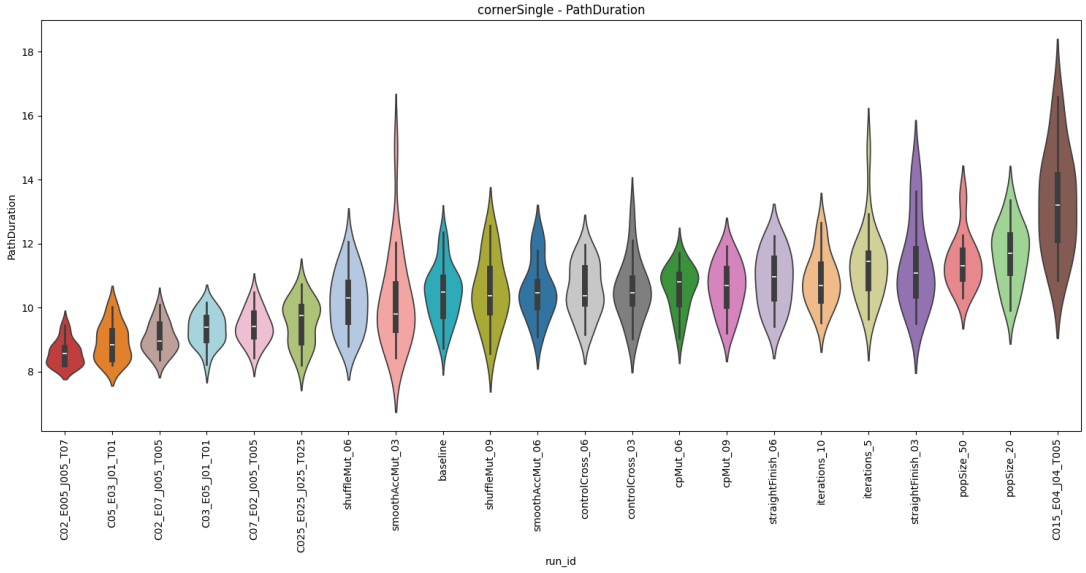


Figure 6.26 Configurations comparison of PathDuration entry in CornerSingle

6.3.6 SmallObstacle - CollisionCount

We can see in Figure 6.21a that our configurations are divided mainly into two groups, those with a mean equal to zero and those with a mean equal to one (and a small group with a mean value equal to 0.5). If we look at Figure 6.27 which shows comparison of *FramesInCollision* entry among configurations, we see that the vast majority of the configurations have a mean value below twenty. This indicates that collisions in this scenario are really insignificant, and most likely similar to what we saw in the *CornerSingle* scenario, they might be just a small corner cut.

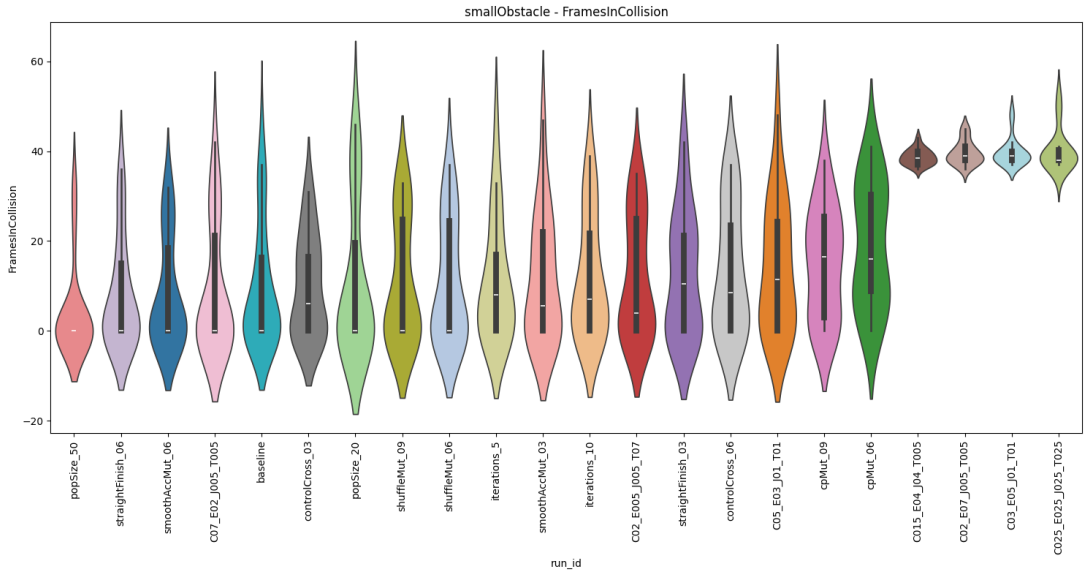


Figure 6.27 Configurations comparison of FramesInCollision entry in SmallObstacle

6.3.7 NarrowCoridorOpposite - GaTimes

Data presented in Figure 6.23a are not surprising. We see that many configurations share the same values, except for the configurations that either lower the population size or the iterations of the GA. This is logical, since both reduce computational time needed for algorithm to finish.

6.3.8 OppositeMultipleAgents - PathJerk

We can see in Figure 6.23a that the configurations share similar data. Unsurprisingly, the best configuration is *C015_E04_J04_T005* since it places the greatest emphasis on the *JerkCost* fitness among other configurations. We can also see that configurations with lower *Collision* fitness weight also have lower values of *JerkCost*. This might be a side effect of not avoiding collisions so often, leading to less sudden changes in velocity. If we look at Figure 6.28 which shows *CollisionCount* in this scenario, we see that these configurations are among the worst performers.

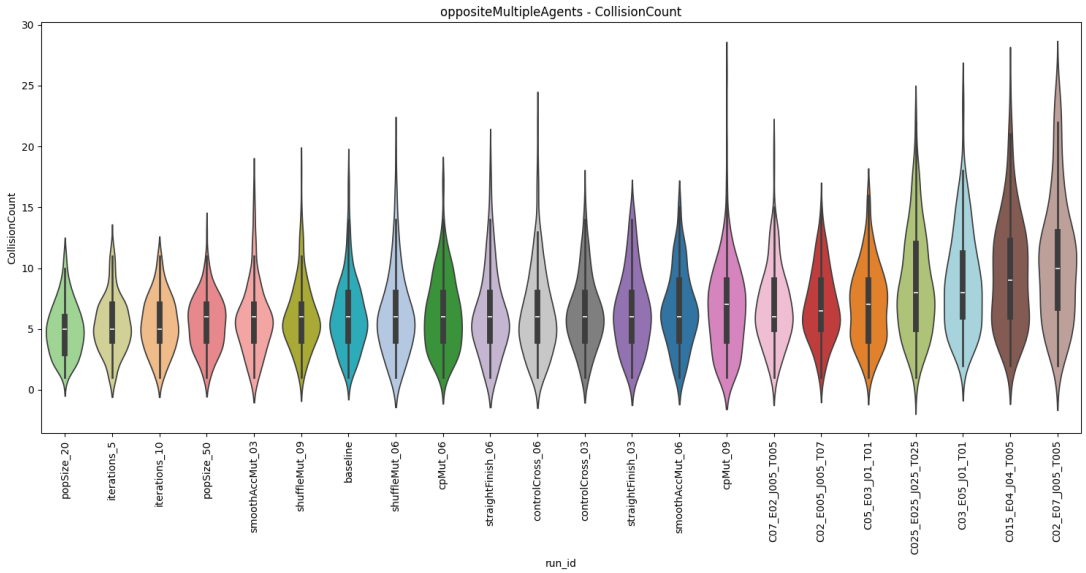


Figure 6.28 Configurations comparison of CollisionCount entry in OppositeMultipleAgents

6.3.9 Additional observations

In this section, we describe some of our additional observations that are not directly visible from the plots. We made these observations primarily while running the baseline configuration.

We start with a common problem in collision avoidance algorithms, that is, agents' oscillations. As explained in Section 2.2.1, this problem is most visible if agents are moving in opposite directions. In our solution, we have not encountered agents oscillating. They might not steer in opposite directions on first try, but they are not repeatedly returning to their previous directions.

The second observation we made was in the *SmallObstacle* scenario, where we noticed that the agent has more human-like behaviour. It starts to notice the obstacle quite early and adjust the steering. Once it passes the point where there is no direct collision with the obstacle, it heads to the destination in a more direct line.

Another observation worth mentioning that we had was in the *NarrowCoridorsOppositeNoNavmeshScenario*. We noticed that some agents are selecting routes that go completely around the narrow corridors, resulting in fewer agents being stuck and less collisions.

6.3.10 Results summary

In this section, we give a brief comparison between the configurations. To do so, we performed a ranking based on the data selected from the experiments. As already mentioned in the previous section 6.2.3, in the configuration comparing graphs, the configurations are sorted according to their mean value. We assign a rank to each configuration based on its order. The configurations with a better mean value (in graphs, those are more to the left) have a lower rank, starting from 1. This gives us a matrix of ranks for each configuration in every scenario-*graph data* combination. We then calculate the average ranking for each configuration to determine their order. The final results are presented in the table in Figure 6.29.

Hyperparameter sweeps' mean ranking	
Hyperparameter sweep	Mean ranking
C07_E02_J005_T005	9.75
C03_E05_J01_T01	9.77
C02_E07_J005_T005	10.0
shuffleMut_06	10.31
shuffleMut_09	10.56
popSize_20	10.77
baseline	10.79
popSize_50	10.88
C025_E025_J025_T025	11.17
C02_E005_J005_T07	11.19
controlCross_03	11.23
C05_E03_J01_T01	11.35
cpMut_06	11.48
cpMut_09	11.67
iterations_10	11.73
smoothAccMut_06	11.79
controlCross_06	11.94
iterations_5	12.23
C015_E04_J04_T005	12.48
smoothAccMut_03	13.06
straightFinish_06	13.65
straightFinish_03	15.21

Figure 6.29 Table showing mean ranking of hyperparameter sweeps, sorted in ascending order.

From the results presented in Figure 6.29 we can see that many configura-

tions perform similarly, but there is a noticeable difference between the first (*C07_E02_J005_T005*) and the last configuration (*straightFinish_03*). Therefore we consider the configuration *C07_E02_J005_T005* to achieve the best results in our experiments and be the overall winner among our configurations.

Due to the size of the ranking matrix, we do not present it here, but it can be found in the form of a table in Attachments B.

7 User documentation

In this chapter, we first describe in Section how to obtain the source codes for our solution and run the application. Then in Section 7.2, we explain how to change the configuration of the genetic algorithm. Lastly, in Section 7.3 we describe how to create new plots.

7.1 Application

You can find the source codes for our solution in the Attachments C in folder *UnityNavigation-BezierIndividual_only*, or directly on github address `https://github.com/lakatop/UnityNavigation/tree/BezierIndividual_only` under the `v1.0.0` tag.

The next thing that our application needs to run is the Unity engine (1) of version 2022.3.7f1.

The complete step-by-step instructions are as follows:

1. Download and install Unity from the following site `https://unity.com/downloadhow-get-started`
2. Create the Unity account if you do not have one, and sign in to Unity Hub.
3. Install Unity Editor via Unity Hub.
4. Add our solution project via the "Add" button. To do so, select the *UnityNavigation-BezierIndividual_only* folder. At this point, you might see little warning in our project in the "Editor version" column, as shown in Figure 7.1. This means that you do not have the correct Unity version. If this is the case, click on the warning button. This should open a new window with the missing version listed as shown in Figure 7.2. Make sure that you have version 2022.3.7f1 selected and click the "Install version 2022.3.7f1" button.
5. Open the project.
6. Open the *StraightLineScene* scene by navigating to the *Scenes* folder inside the editor and double clicking on the *StraightLineScene* file.
7. Check that inside the *Jobs* menu you have *Enable Compilation* checked and everything else unchecked. This will enable parallelism.
8. Optionally, turn on Gizmos to see the debug view.

9. Click on the play button.

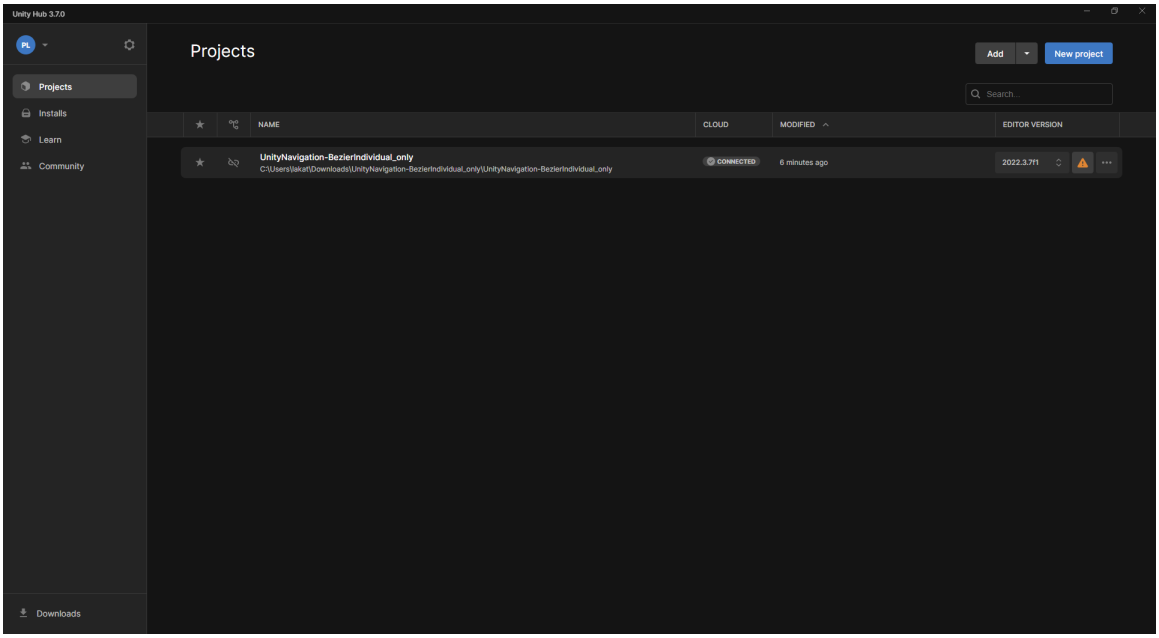


Figure 7.1 Incorrect Unity Editor version warning

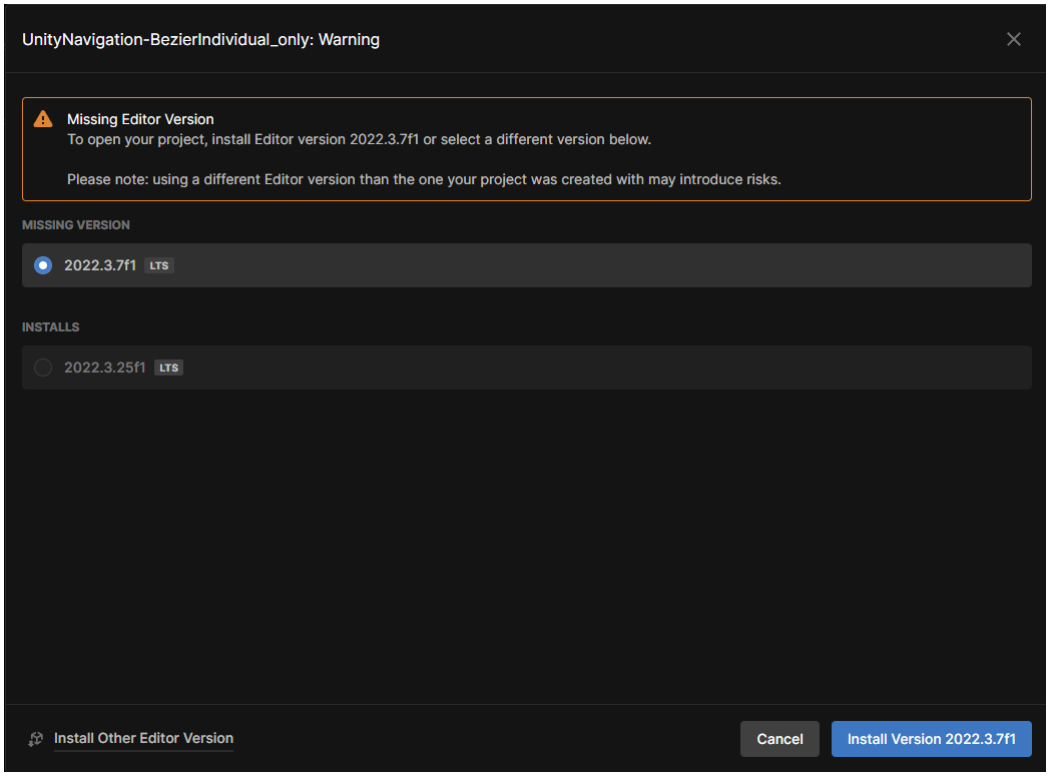


Figure 7.2 Warning window to install correct version of Unity Editor

After completing the instructions above, you should see simulation running and the first StraightLineScene scenario is started. The simulation will progressively run each scenario in the same order in which they are defined in Section 6.1.

Each scenario will run once, and after the last scenario, the application will stop. The current configuration of the genetic algorithm is the baseline (defined in Section 6.2.1). We explain how to change the configuration in Section 7.2.

7.2 Configuration change

To change the configuration from the *baseline*, open the *Director.cs* file inside the *GeneticAlgorithm* folder from the Attachments C.

In the table below in Figure 7.3, we list the mapping between the hyperparameters and their code variables.

Mapping of the hyperparameters to their code variables	
Hyperparameter	Code variable
Population size	`populationSize`
Number of iterations of the GA	`iterations`
StraightFinish mutation probability	`mutationProb` inside `ga.straightFinishMutation` struct
ShuffleAcc mutation probability	`mutationProb` inside `ga.shuffleMutation` struct
SmoothAcc mutation probability	`mutationProb` inside `ga.smoothMutation` struct
ShuffleControlPoints mutation probability	`mutationProb` inside `ga.controlPointsMutation` struct
Collision fitness weight	`weight` inside `ga.collisionFitness` struct
EndDistance fitness weight	`weight` inside `ga.endDistanceFitness` struct
JerkCost fitness weight	`weight` inside `ga.jerkFitness` struct
TimeToDestination fitness weight	`weight` inside `ga.ttdFitness` struct

Figure 7.3 Table of mappings between hyperparameters and their code variables

If you want to change the scenario runs, open the *SimulationManager* inside the *Managers* folder from the Attachments C. Locate the `CreateScearios` method and change the values in the scenario constructor accordingly.

After changing the configuration, save the file edits and open the application. It should automatically rebuild with the new changes. After the rebuild, click on the play button, and you should see the simulation running with the new configuration.

7.3 Plotting the graphs

You can see the already created plots inside the *Plotting/Runs* folder in the Attachments C. The folders with plots are following:

- *ConfigComparePlots* - contains plots comparing every configuration.
- *DependentComparePlots* - contains plots comparing dependent hyperparameters configurations.

- *NonDependentComparePlots* - containt plots comparing nondependent hyperparameters configurations.
- *ScenarioPlots* - contains a folder for each configuration with the corresponding plots for each scenario.

For graph creation, we renamed the folders according to the mappings described in Section 6.2. This mapping is also captured in the *run_names.txt* file inside the *Plotting* folder in the Attachments C.

To create new graphs, you can run the plotting script more closely described in Section 6.2.3. You also need to make sure that your *Plotting/Runs* folder contains only folders with configuration logs (without folders of already generated plots).

8 Future work

In this chapter we present the parts of our solution that could be reiterated and improved on and what could be the future focus.

8.1 Differential evolution

Differential evolution is a method first introduced by Storn and Price (13) and is used for global optimisation in continuous space. As presented in the study (13), "the basic strategy employs the difference of two randomly selected parameter vectors as the source of random variations for a third parameter vector." Where vectors are perceived as our individuals.

This method is highly effective and could potentially also be used to find the best individual in our solution. It would be interesting to try to incorporate the differential evolution into the next iterations of our solution.

8.2 Preferred velocity

In some of the modern collision avoidance algorithms, there is a possibility to define the preferred velocity of an agent. This is useful mainly in cases with multiple interdestinations where we do not want an agent to stop completely. This possibility would be a good later addition to our solution.

8.3 Other individual representation

The Bezier curves are not flawless. The main problem is that if we want to determine position on a spline, we need to use a piecewise linear approximation. This comes off as a bit cumbersome and might also have some performance impact. It might be a good idea to explore some other individual encodings using different types of splines with more appropriate attributes.

8.4 Tuning of hyperparameters

Based on the results of our experiments in Section 6.2.3, the next step would be to use these results and create a new baseline solution with more tuned hyperparameters to maximise correctness. Probably the best method would be to use Pareto optimisation and find a Pareto-optimal configuration.

Conclusion

The overall result of this thesis is a functional genetic algorithm that serves as a local space search algorithm to explore viable paths that successfully navigate an agent to its destination, while taking into account all the quality path metrics defined in Section 1.2.

Now, we go through the defined requirements of this thesis and describe how we completed them:

(R1) (*Implement collision avoidance algorithm using genetic algorithm.*) – fulfilling this objective is described throughout Chapter 4, where we describe implementation details, and Chapter 5, where we describe solution overview of our genetic algorithm.

(R2) (*Provide multiple scenarios and perform experiments to test the implemented collision avoidance algorithm.*) – meeting this objective is describe in Chapter 6, where we present experiments that were run to test our solution.

(R3) (*Run experiments on multiple configurations of the implemented genetic algorithm.*) – achieving this requirement is described in Section 6.2, where we describe multiple sweeps performed on our algorithm which created new configurations.

(R4) (*The metrics defined for the evaluation of the path are captured and saved from the experiments.*) – fulfilling this objective is described in Section 6.2.3 where we describe logging performed by our solution.

(R5) (*The experiments are visually observable in simulation.*) – meeting this objective is described in Section 6.1 where we present scenarios prepared for the experiments, as well as in Chapter 7 where we describe how to setup and run simulation that contains these scenarios.

(R6) (*The results of the experiments are captured, analysed, and discussed in the form of graphs.*) – fulfilling this requirement is described in Section 6.2.3 where we present plots created from data captured in experiments, and Section 6.3 where we analyse and discuss these results.

Bibliography

[1] Unity Engine. https://unity.com/.

[2] Unity Engine - Jobs system. https://docs.unity3d.com/Manual/JobSystem.html.

[3] Unity Engine - Burst compiler. https://docs.unity3d.com/Packages/com.unity.burst@0.2/manual/index.html.

[4] Unity Engine - Collections package . https://docs.unity3d.com/Packages/com.unity.collections@2.4/manual/index.html.

[5] Unity Engine - Scenes. https://docs.unity3d.com/Manual/CreatingScenes.html.

[6] Unity Engine - Time property - realtimeSinceStartupAsDouble. https://docs.unity3d.com/ScriptReference/Time-realtimeSinceStartupAsDouble.html.

[7] Github repository - NativeQuadTree . https://github.com/marijnz/NativeQuadtree.

[8] Python library - Pandas . https://pandas.pydata.org/.

[9] Python library - Matplotlib . https://matplotlib.org/.

[10] Python library - Seaborn . https://seaborn.pydata.org/index.html.

[11] Jupyter notebook . https://jupyter.org/.

[12] Unity Engine - Scale and units . https://docs.unity3d.com/2019.2/Documentation/Manual/BestPracticeMakingBelievableVisuals1.html.

[13] Storn, Rainer; Price, Kenneth (1995). "Differential evolution — a simple and efficient scheme for global optimization over continuous spaces". https://cse.engineering.nyu.edu/~mleung/CS909/s04/Storn95-012.pdf.

[14] Reza Entezari-Maleki, Ali Movaghar. "A Genetic Algorithm to Increase the Throughput of the Computational Grids". http://sina.sharif.ac.ir/~movaghar/IJGDC_Entezari.pdf.

[15] Diego Perez, Spyridon Samothrakis, Simon Lucas, and Philipp Rohlfshagen (2013). "Rolling horizon evolution versus tree search for navigation in single-player real-time games". https://dl.acm.org/doi/10.1145/2463372.2463413.

[16] Fiorini, Paolo; Shiller, Zvi (1998). "Motion Planning in Dynamic Environments Using Velocity Obstacles". https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d8315aaef1544046a184f7ad8252cf1de0def800.

[17] J. Van Den Berg, M. Lin, D. Manocha (2008). "Reciprocal velocity obstacles for real-time multi-agent navigation". https://gamma.cs.unc.edu/RVO/icra2008.pdf.

[18] Vesentini, Federico; Muradore, Riccardo; Fiorini, Paolo. (2024). "A survey on Velocity Obstacle paradigm".. https://www.researchgate.net/publication/377989774_A_survey_on_Velocity_Obstacle_paradigm.

[19] J. Van Den Berg, S.J. Guy, M. Lin, D. Manocha (2009). "Reciprocal n-body collision avoidance". https://gamma.cs.unc.edu/ORCA/publications/ORCA.pdf.

[20] Rabin, Steve (2017). "Game AI Pro 3: Collected Wisdom of Game AI Professionals". https://www.gameaipro.com/GameAIPro3/GameAIPro3_Chapter19_RVO_and_ORCA_How_They_Really_Work.pdf.

[21] Nallaperuma, Sam; Neumann, Frank; Bonyadi, Mohammad reza; Michalewicz, Zbigniew. (2014). "EVOR : An Online Evolutionary Algorithm for Car Racing Games". https://www.researchgate.net/publication/261361821_EVOR_An_Online_Evolutionary_Algorithm_for_Car_Racing_Games.

[22] Atlassian - A complete guide to violin plots. https://www.atlassian.com/data/charts/violin-plot-complete-guide.

List of Figures

List of Tables

List of Abbreviations

VO	Velocity Obstacles
RVO	Reciprocal velocity Obstacles
ORCA	Optimal reciprocal collision avoidance
GA	Genetic algorithm

A Unity dictionary

In this chapter, we list Unity-related terms that we use in the thesis. Their explanation can be found on the Editor Manual page `https://docs.unity3d.com/2022.3/Documentation/Manual/`, or on the Unity Scripting Reference page `https://docs.unity3d.com/ScriptReference/`.

- NavMesh – a mesh created by Unity to define walkable areas and obstacles to perform path finding.
- NavMeshSurface – component for building and enabling a NavMesh surface for one agent type.
- NavMeshModifier – modifier for affecting the NavMesh generation.
- NavMeshAgent – component attached to unit to allow navigation using NavMesh.
- Walkable property – property of `NavMeshModifier`, setting area type to walkable.
- Not walkable property – property of `NavMeshModifier`, setting area type to not walkable.
- Scene – set of assets that contain game or application.
- Time – interface to get information about time from Unity.

B Hyperparameter sweeps' ranking

In this chapter, we present all hyperparameter sweeps' rankings across all scenarios. It is captured in table B.1.

Table B.1 Hyperparameter sweeps' rankings across all scenarios. Sorted in ascending order across columns based on their mean value.

(a) Part 1

Sweeps ranking across scenarios			
Scenario names / Mean	Sweeps values		
-	C07_E02_J005_T005	C03_E05_J01_T01	C02_E07_J005_T005
Mean	9.75	9.771	10.0
straightLine - PathLength	6	5	2
straightLine - PathDuration	3	5	2
straightLine - CollisionCount	15	17	18
straightLine - FramesInCollision	15	17	18
straightLine - PathJerk	4	9	21
straightLine - GaTimes	5	21	19
smallObstacle - PathLength	8	3	1
smallObstacle - PathDuration	6	4	2
smallObstacle - CollisionCount	2	19	20
smallObstacle - FramesInCollision	4	21	20
smallObstacle - PathJerk	10	3	2
smallObstacle - GaTimes	5	21	18
oppositeMultipleAgents - PathLength	10	1	2
oppositeMultipleAgents - PathDuration	6	2	1
oppositeMultipleAgents - CollisionCount	16	20	22
oppositeMultipleAgents - FramesInCollision	18	21	22
oppositeMultipleAgents - PathJerk	16	2	3
oppositeMultipleAgents - GaTimes	7	10	21
oppositeCircleAgents - PathLength	6	2	1
oppositeCircleAgents - PathDuration	6	1	2
oppositeCircleAgents - CollisionCount	7	20	21
oppositeCircleAgents - FramesInCollision	6	19	21
oppositeCircleAgents - PathJerk	19	22	21
oppositeCircleAgents - GaTimes	15	16	22
oppositeAgents - PathLength	9	2	4
oppositeAgents - PathDuration	5	2	4
oppositeAgents - CollisionCount	19	21	22
oppositeAgents - FramesInCollision	18	21	22
oppositeAgents - PathJerk	12	15	5
oppositeAgents - GaTimes	5	22	19
narrowCoridorsOppositeNoNavmeshScenario - PathLength	17	2	1
narrowCoridorsOppositeNoNavmeshScenario - PathDuration	9	3	1
narrowCoridorsOppositeNoNavmeshScenario - CollisionCount	21	5	3
narrowCoridorsOppositeNoNavmeshScenario - FramesInCollision	22	2	1
narrowCoridorsOppositeNoNavmeshScenario - PathJerk	11	3	1
narrowCoridorsOppositeNoNavmeshScenario - GaTimes	6	18	21
narrowCoridorOpposite - PathLength	14	1	2
narrowCoridorOpposite - PathDuration	7	1	2
narrowCoridorOpposite - CollisionCount	5	2	1
narrowCoridorOpposite - FramesInCollision	6	3	1
narrowCoridorOpposite - PathJerk	6	2	3
narrowCoridorOpposite - GaTimes	9	8	6
cornerSingle - PathLength	5	4	3
cornerSingle - PathDuration	5	4	3
cornerSingle - CollisionCount	13	3	15
cornerSingle - FramesInCollision	3	6	2
cornerSingle - PathJerk	21	17	18
cornerSingle - GaTimes	5	21	18

(b) Part 2

Sweeps ranking across scenarios			
Scenario names / Mean	Sweeps values		
-	shuffleMut_06	shuffleMut_09	popSize_20
Mean	10.312	10.562	10.771
straightLine - PathLength	10	4	17
straightLine - PathDuration	9	6	12
straightLine - CollisionCount	5	4	7
straightLine - FramesInCollision	5	4	7
straightLine - PathJerk	6	3	13
straightLine - GaTimes	12	16	1
smallObstacle - PathLength	9	6	20
smallObstacle - PathDuration	8	7	19
smallObstacle - CollisionCount	7	6	3
smallObstacle - FramesInCollision	9	8	7
smallObstacle - PathJerk	8	14	17
smallObstacle - GaTimes	14	19	1
oppositeMultipleAgents - PathLength	7	6	20
oppositeMultipleAgents - PathDuration	8	7	20
oppositeMultipleAgents - CollisionCount	8	6	1
oppositeMultipleAgents - FramesInCollision	13	10	5
oppositeMultipleAgents - PathJerk	21	18	6
oppositeMultipleAgents - GaTimes	20	16	1
oppositeCircleAgents - PathLength	9	8	22
oppositeCircleAgents - PathDuration	9	7	22
oppositeCircleAgents - CollisionCount	11	9	1
oppositeCircleAgents - FramesInCollision	11	8	4
oppositeCircleAgents - PathJerk	12	14	1
oppositeCircleAgents - GaTimes	20	14	1
oppositeAgents - PathLength	7	6	20
oppositeAgents - PathDuration	6	8	18
oppositeAgents - CollisionCount	3	10	11
oppositeAgents - FramesInCollision	5	9	15
oppositeAgents - PathJerk	4	18	20
oppositeAgents - GaTimes	16	14	1
narrowCoridorsOppositeNoNavmeshScenario - PathLength	7	5	14
narrowCoridorsOppositeNoNavmeshScenario - PathDuration	15	6	7
narrowCoridorsOppositeNoNavmeshScenario - CollisionCount	9	15	1
narrowCoridorsOppositeNoNavmeshScenario - FramesInCollision	8	11	5
narrowCoridorsOppositeNoNavmeshScenario - PathJerk	5	7	20
narrowCoridorsOppositeNoNavmeshScenario - GaTimes	15	17	1
narrowCoridorOpposite - PathLength	7	12	20
narrowCoridorOpposite - PathDuration	8	9	18
narrowCoridorOpposite - CollisionCount	18	21	8
narrowCoridorOpposite - FramesInCollision	10	20	15
narrowCoridorOpposite - PathJerk	12	17	19
narrowCoridorOpposite - GaTimes	17	18	1
cornerSingle - PathLength	8	11	21
cornerSingle - PathDuration	7	10	21
cornerSingle - CollisionCount	20	2	7
cornerSingle - FramesInCollision	11	20	22
cornerSingle - PathJerk	9	2	3
cornerSingle - GaTimes	17	19	1

(c) Part 3

Sweeps ranking across scenarios			
Scenario names / Mean	Sweeps values		
-	baseline	popSize_50	C025_E025_J025_T025
Mean	10.792	10.875	11.167
straightLine - PathLength	16	18	3
straightLine - PathDuration	16	18	4
straightLine - CollisionCount	14	6	20
straightLine - FramesInCollision	14	6	20
straightLine - PathJerk	5	16	8
straightLine - GaTimes	7	3	22
smallObstacle - PathLength	15	17	5
smallObstacle - PathDuration	15	14	5
smallObstacle - CollisionCount	8	1	22
smallObstacle - FramesInCollision	5	1	22
smallObstacle - PathJerk	6	19	21
smallObstacle - GaTimes	7	3	20
oppositeMultipleAgents - PathLength	14	19	4
oppositeMultipleAgents - PathDuration	13	17	4
oppositeMultipleAgents - CollisionCount	7	4	19
oppositeMultipleAgents - FramesInCollision	9	6	17
oppositeMultipleAgents - PathJerk	14	7	4
oppositeMultipleAgents - GaTimes	8	3	22
oppositeCircleAgents - PathLength	13	19	5
oppositeCircleAgents - PathDuration	14	18	5
oppositeCircleAgents - CollisionCount	15	2	14
oppositeCircleAgents - FramesInCollision	13	2	5
oppositeCircleAgents - PathJerk	7	3	17
oppositeCircleAgents - GaTimes	7	4	21
oppositeAgents - PathLength	17	15	5
oppositeAgents - PathDuration	15	10	7
oppositeAgents - CollisionCount	9	7	18
oppositeAgents - FramesInCollision	7	4	19
oppositeAgents - PathJerk	13	21	6
oppositeAgents - GaTimes	7	3	12
narrowCoridorsOppositeNoNavmeshScenario - PathLength	11	13	3
narrowCoridorsOppositeNoNavmeshScenario - PathDuration	10	8	2
narrowCoridorsOppositeNoNavmeshScenario - CollisionCount	13	2	20
narrowCoridorsOppositeNoNavmeshScenario - FramesInCollision	10	3	14
narrowCoridorsOppositeNoNavmeshScenario - PathJerk	13	21	8
narrowCoridorsOppositeNoNavmeshScenario - GaTimes	7	3	22
narrowCoridorOpposite - PathLength	16	22	3
narrowCoridorOpposite - PathDuration	14	20	3
narrowCoridorOpposite - CollisionCount	12	14	4
narrowCoridorOpposite - FramesInCollision	12	22	4
narrowCoridorOpposite - PathJerk	10	20	4
narrowCoridorOpposite - GaTimes	7	3	13
cornerSingle - PathLength	10	20	6
cornerSingle - PathDuration	9	20	6
cornerSingle - CollisionCount	6	16	1
cornerSingle - FramesInCollision	8	14	5
cornerSingle - PathJerk	13	12	22
cornerSingle - GaTimes	7	3	20

(d) Part 4

Sweeps ranking across scenarios			
Scenario names / Mean	Sweeps values		
-	C02_E005_J005_T07	controlCross_03	C05_E03_J01_T01
Mean	11.188	11.229	11.354
straightLine - PathLength	1	8	7
straightLine - PathDuration	1	10	7
straightLine - CollisionCount	19	13	16
straightLine - FramesInCollision	19	13	16
straightLine - PathJerk	22	2	19
straightLine - GaTimes	18	14	20
smallObstacle - PathLength	4	7	2
smallObstacle - PathDuration	1	9	3
smallObstacle - CollisionCount	12	10	15
smallObstacle - FramesInCollision	13	6	16
smallObstacle - PathJerk	18	13	11
smallObstacle - GaTimes	15	16	22
oppositeMultipleAgents - PathLength	5	11	3
oppositeMultipleAgents - PathDuration	5	10	3
oppositeMultipleAgents - CollisionCount	17	12	18
oppositeMultipleAgents - FramesInCollision	4	3	19
oppositeMultipleAgents - PathJerk	22	13	5
oppositeMultipleAgents - GaTimes	9	17	11
oppositeCircleAgents - PathLength	4	11	3
oppositeCircleAgents - PathDuration	4	10	3
oppositeCircleAgents - CollisionCount	4	16	17
oppositeCircleAgents - FramesInCollision	1	17	10
oppositeCircleAgents - PathJerk	20	11	18
oppositeCircleAgents - GaTimes	19	11	18
oppositeAgents - PathLength	3	8	1
oppositeAgents - PathDuration	1	9	3
oppositeAgents - CollisionCount	6	12	16
oppositeAgents - FramesInCollision	8	13	12
oppositeAgents - PathJerk	22	8	7
oppositeAgents - GaTimes	17	13	20
narrowCoridorsOppositeNoNavmeshScenario - PathLength	16	10	4
narrowCoridorsOppositeNoNavmeshScenario - PathDuration	20	13	4
narrowCoridorsOppositeNoNavmeshScenario - CollisionCount	12	18	22
narrowCoridorsOppositeNoNavmeshScenario - FramesInCollision	17	19	21
narrowCoridorsOppositeNoNavmeshScenario - PathJerk	22	4	15
narrowCoridorsOppositeNoNavmeshScenario - GaTimes	8	12	19
narrowCoridorOpposite - PathLength	5	8	4
narrowCoridorOpposite - PathDuration	4	12	5
narrowCoridorOpposite - CollisionCount	19	17	13
narrowCoridorOpposite - FramesInCollision	14	13	9
narrowCoridorOpposite - PathJerk	22	11	8
narrowCoridorOpposite - GaTimes	11	16	20
cornerSingle - PathLength	1	13	2
cornerSingle - PathDuration	1	13	2
cornerSingle - CollisionCount	22	8	10
cornerSingle - FramesInCollision	1	9	4
cornerSingle - PathJerk	19	4	20
cornerSingle - GaTimes	9	13	22

(e) Part 5

Sweeps ranking across scenarios			
Scenario names / Mean	Sweeps values		
-	cpMut_06	cpMut_09	iterations_10
Mean	11.479	11.667	11.729
straightLine - PathLength	14	9	19
straightLine - PathDuration	11	8	19
straightLine - CollisionCount	11	10	9
straightLine - FramesInCollision	11	10	9
straightLine - PathJerk	17	14	15
straightLine - GaTimes	6	17	4
smallObstacle - PathLength	19	10	18
smallObstacle - PathDuration	20	10	18
smallObstacle - CollisionCount	17	18	13
smallObstacle - FramesInCollision	18	17	12
smallObstacle - PathJerk	20	4	5
smallObstacle - GaTimes	6	17	4
oppositeMultipleAgents - PathLength	16	8	17
oppositeMultipleAgents - PathDuration	12	11	14
oppositeMultipleAgents - CollisionCount	9	15	3
oppositeMultipleAgents - FramesInCollision	2	1	8
oppositeMultipleAgents - PathJerk	11	9	12
oppositeMultipleAgents - GaTimes	6	19	4
oppositeCircleAgents - PathLength	14	12	18
oppositeCircleAgents - PathDuration	13	12	16
oppositeCircleAgents - CollisionCount	19	18	8
oppositeCircleAgents - FramesInCollision	20	18	14
oppositeCircleAgents - PathJerk	5	6	8
oppositeCircleAgents - GaTimes	5	10	3
oppositeAgents - PathLength	18	14	13
oppositeAgents - PathDuration	13	16	11
oppositeAgents - CollisionCount	1	8	14
oppositeAgents - FramesInCollision	1	6	10
oppositeAgents - PathJerk	9	10	19
oppositeAgents - GaTimes	6	11	4
narrowCoridorsOppositeNoNavmeshScenario - PathLength	19	8	18
narrowCoridorsOppositeNoNavmeshScenario - PathDuration	19	18	16
narrowCoridorsOppositeNoNavmeshScenario - CollisionCount	14	19	7
narrowCoridorsOppositeNoNavmeshScenario - FramesInCollision	13	18	16
narrowCoridorsOppositeNoNavmeshScenario - PathJerk	9	10	17
narrowCoridorsOppositeNoNavmeshScenario - GaTimes	5	16	4
narrowCoridorOpposite - PathLength	11	6	18
narrowCoridorOpposite - PathDuration	11	6	16
narrowCoridorOpposite - CollisionCount	9	6	10
narrowCoridorOpposite - FramesInCollision	7	5	18
narrowCoridorOpposite - PathJerk	7	5	14
narrowCoridorOpposite - GaTimes	12	22	4
cornerSingle - PathLength	15	14	17
cornerSingle - PathDuration	14	15	17
cornerSingle - CollisionCount	9	11	4
cornerSingle - FramesInCollision	10	7	16
cornerSingle - PathJerk	11	14	6
cornerSingle - GaTimes	6	12	4

(f) Part 6

Sweeps ranking across scenarios			
Scenario names / Mean	Sweeps values		
-	smoothAccMut_06	controlCross_06	iterations_5
Mean	11.792	11.938	12.229
straightLine - PathLength	13	11	20
straightLine - PathDuration	15	13	20
straightLine - CollisionCount	2	12	8
straightLine - FramesInCollision	2	12	8
straightLine - PathJerk	7	11	10
straightLine - GaTimes	13	15	2
smallObstacle - PathLength	11	12	21
smallObstacle - PathDuration	11	12	21
smallObstacle - CollisionCount	4	16	14
smallObstacle - FramesInCollision	3	15	10
smallObstacle - PathJerk	12	7	9
smallObstacle - GaTimes	12	13	2
oppositeMultipleAgents - PathLength	12	9	22
oppositeMultipleAgents - PathDuration	15	9	19
oppositeMultipleAgents - CollisionCount	14	11	2
oppositeMultipleAgents - FramesInCollision	12	7	16
oppositeMultipleAgents - PathJerk	17	8	15
oppositeMultipleAgents - GaTimes	15	18	2
oppositeCircleAgents - PathLength	16	10	20
oppositeCircleAgents - PathDuration	19	11	20
oppositeCircleAgents - CollisionCount	10	13	5
oppositeCircleAgents - FramesInCollision	7	12	16
oppositeCircleAgents - PathJerk	13	4	9
oppositeCircleAgents - GaTimes	17	12	2
oppositeAgents - PathLength	10	12	21
oppositeAgents - PathDuration	19	12	17
oppositeAgents - CollisionCount	5	13	2
oppositeAgents - FramesInCollision	11	14	2
oppositeAgents - PathJerk	17	2	14
oppositeAgents - GaTimes	15	18	2
narrowCoridorsOppositeNoNavmeshScenario - PathLength	12	9	20
narrowCoridorsOppositeNoNavmeshScenario - PathDuration	11	14	12
narrowCoridorsOppositeNoNavmeshScenario - CollisionCount	8	17	4
narrowCoridorsOppositeNoNavmeshScenario - FramesInCollision	7	15	12
narrowCoridorsOppositeNoNavmeshScenario - PathJerk	14	6	18
narrowCoridorsOppositeNoNavmeshScenario - GaTimes	14	20	2
narrowCoridorOpposite - PathLength	9	10	21
narrowCoridorOpposite - PathDuration	13	10	17
narrowCoridorOpposite - CollisionCount	11	15	7
narrowCoridorOpposite - FramesInCollision	8	11	21
narrowCoridorOpposite - PathJerk	15	9	18
narrowCoridorOpposite - GaTimes	14	19	2
cornerSingle - PathLength	9	12	19
cornerSingle - PathDuration	11	12	18
cornerSingle - CollisionCount	12	14	17
cornerSingle - FramesInCollision	19	13	21
cornerSingle - PathJerk	15	7	5
cornerSingle - GaTimes	15	16	2

(g) Part 7

Sweeps ranking across scenarios			
Scenario names / Mean	Sweeps values		
-	C015_E04_J04_T005	smoothAccMut_03	straightFinish_06
Mean	12.479	13.062	13.646
straightLine - PathLength	22	12	15
straightLine - PathDuration	22	14	17
straightLine - CollisionCount	1	3	22
straightLine - FramesInCollision	1	3	22
straightLine - PathJerk	1	18	20
straightLine - GaTimes	10	9	11
smallObstacle - PathLength	22	14	13
smallObstacle - PathDuration	22	16	13
smallObstacle - CollisionCount	21	9	5
smallObstacle - FramesInCollision	19	11	2
smallObstacle - PathJerk	1	16	15
smallObstacle - GaTimes	8	10	11
oppositeMultipleAgents - PathLength	21	13	15
oppositeMultipleAgents - PathDuration	22	18	16
oppositeMultipleAgents - CollisionCount	21	5	10
oppositeMultipleAgents - FramesInCollision	20	14	11
oppositeMultipleAgents - PathJerk	1	10	20
oppositeMultipleAgents - GaTimes	5	14	13
oppositeCircleAgents - PathLength	7	21	15
oppositeCircleAgents - PathDuration	8	21	15
oppositeCircleAgents - CollisionCount	22	3	12
oppositeCircleAgents - FramesInCollision	22	3	15
oppositeCircleAgents - PathJerk	2	16	10
oppositeCircleAgents - GaTimes	6	13	8
oppositeAgents - PathLength	22	16	11
oppositeAgents - PathDuration	22	20	14
oppositeAgents - CollisionCount	20	17	15
oppositeAgents - FramesInCollision	20	17	16
oppositeAgents - PathJerk	1	11	3
oppositeAgents - GaTimes	21	9	8
narrowCoridorsOppositeNoNavmeshScenario - PathLength	6	22	15
narrowCoridorsOppositeNoNavmeshScenario - PathDuration	5	22	17
narrowCoridorsOppositeNoNavmeshScenario - CollisionCount	10	6	16
narrowCoridorsOppositeNoNavmeshScenario - FramesInCollision	4	6	20
narrowCoridorsOppositeNoNavmeshScenario - PathJerk	2	12	16
narrowCoridorsOppositeNoNavmeshScenario - GaTimes	13	11	10
narrowCoridorOpposite - PathLength	19	15	13
narrowCoridorOpposite - PathDuration	22	21	15
narrowCoridorOpposite - CollisionCount	3	20	16
narrowCoridorOpposite - FramesInCollision	2	16	17
narrowCoridorOpposite - PathJerk	1	16	13
narrowCoridorOpposite - GaTimes	5	15	21
cornerSingle - PathLength	22	7	16
cornerSingle - PathDuration	22	8	16
cornerSingle - CollisionCount	18	21	5
cornerSingle - FramesInCollision	17	12	18
cornerSingle - PathJerk	1	10	8
cornerSingle - GaTimes	14	11	10

(h) Part 7

Sweeps ranking across scenarios	
Scenario names / Mean	Sweeps values
-	straightFinish_03
Mean	15.208
straightLine - PathLength	21
straightLine - PathDuration	21
straightLine - CollisionCount	21
straightLine - FramesInCollision	21
straightLine - PathJerk	12
straightLine - GaTimes	8
smallObstacle - PathLength	16
smallObstacle - PathDuration	17
smallObstacle - CollisionCount	11
smallObstacle - FramesInCollision	14
smallObstacle - PathJerk	22
smallObstacle - GaTimes	9
oppositeMultipleAgents - PathLength	18
oppositeMultipleAgents - PathDuration	21
oppositeMultipleAgents - CollisionCount	13
oppositeMultipleAgents - FramesInCollision	15
oppositeMultipleAgents - PathJerk	19
oppositeMultipleAgents - GaTimes	12
oppositeCircleAgents - PathLength	17
oppositeCircleAgents - PathDuration	17
oppositeCircleAgents - CollisionCount	6
oppositeCircleAgents - FramesInCollision	9
oppositeCircleAgents - PathJerk	15
oppositeCircleAgents - GaTimes	9
oppositeAgents - PathLength	19
oppositeAgents - PathDuration	21
oppositeAgents - CollisionCount	4
oppositeAgents - FramesInCollision	3
oppositeAgents - PathJerk	16
oppositeAgents - GaTimes	10
narrowCoridorsOppositeNoNavmeshScenario - PathLength	21
narrowCoridorsOppositeNoNavmeshScenario - PathDuration	21
narrowCoridorsOppositeNoNavmeshScenario - CollisionCount	11
narrowCoridorsOppositeNoNavmeshScenario - FramesInCollision	9
narrowCoridorsOppositeNoNavmeshScenario - PathJerk	19
narrowCoridorsOppositeNoNavmeshScenario - GaTimes	9
narrowCoridorOpposite - PathLength	17
narrowCoridorOpposite - PathDuration	19
narrowCoridorOpposite - CollisionCount	22
narrowCoridorOpposite - FramesInCollision	19
narrowCoridorOpposite - PathJerk	21
narrowCoridorOpposite - GaTimes	10
cornerSingle - PathLength	18
cornerSingle - PathDuration	19
cornerSingle - CollisionCount	19
cornerSingle - FramesInCollision	15
cornerSingle - PathJerk	16
cornerSingle - GaTimes	8

C Attachments

```
.
├── UnityNavigation-BezierIndividual_only
│   ├── Assets
│   │   └── Scripts - folder containing source codes to the solution
│   ├── Docs - folder containing documentation generated by doxygen
│   ├── Packages- Unity packages
│   ├── Plotting
│   │   ├── plotting.ipynb - script to generate plots
│   │   ├── Runs - folder containing logs and generated plots
│   │   └── runs_names.txt - file containing mapping between folders and
│   │       their corresponding configurations
│   └── ProjectSettings - Unity project settings
├── tex - source codes of the thesis text in LaTeX
└── thesis.pdf - text of the thesis
```