2018

Evolutionary Optimization for Safe Navigation of an Autonomous Robot in Cluttered Dynamic Unknown Environments

Arash Roshanineshat
Cleveland State University

Evolutionary Optimization for Safe Navigation of an Autonomous Robot in Cluttered Dynamic Unknown Environments

ARASH ROSHANINESHAT

Bachelor of Science in Electrical Engineering

University of Zanjan

June 2015

submitted in partial fulfillment of requirements for the degree

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

at the

CLEVELAND STATE UNIVERSITY

August 2018

We hereby approve this thesis for

ARASH ROSHANINESHAT

Candidate for the Master of Science in Electrical Engineering degree for the

Department of Electrical Engineering and Computer Science

and the CLEVELAND STATE UNIVERSITY'S

College of Graduate Studies by

Thesis Chairperson, Dr. Dan Simon

Department & Date

Thesis Committee Member, Dr. Lili Dong

Department & Date

Thesis Committee Member, Dr. Mohammad Shokrolah Shirazi

Department & Date

Student's Date of Defense: June 18, 2018

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Prof. Dan Simon, for his continuous, invaluable guidance and patience in my work. His motivation, creative problem-solving skills and immense knowledge were gifts that I hope I can embody. I thank Taylor Barto, Saman Khademi, Seyed Fakoorian, Haniye Mohammadi, Donald Ebeigbe and Mohamed Abdelhady for their help and support throughout my project and research as lab mates, and for being there when I needed them. I also would like to thank Dr. Jonathan Weintroub for mentoring my internship at Harvard-Smithsonian, and Dr. Richard Prestage for advising my internship at National Radio Astronomy Observatory. I appreciate their encouragement and support for my projects, which helped me develop programming and problem-solving skills. I want to thank Saba, Leili, Ashkan and my friends at Cleveland State University for providing joy and laughter at the right time when I needed them. I thank the people at the IEEE section of Cleveland State University who let me be involved in several robotic and volunteering opportunities. Last but most of all, I would like to thank my parents and my sister, who always tried to sacrificially help and support me through the years and who were there for me even though I am thousands of miles away.

Evolutionary Optimization for Safe Navigation of an Autonomous Robot in Cluttered Dynamic Unknown Environments

ARASH ROSHANINESHAT

ABSTRACT

We present a path planning approach based on probabilistic methods for a robot to navigate in a cluttered, dynamic, unknown environment. There are dynamic obstacles moving around and static obstacles located in the map. The robot does not have any prior information about them but should be able to navigate through the map beginning from a known starting point and safely ending at a known target point. The only information the robot has is the location of the starting point and the target point and it uses sensory information to collect information about its surroundings. Our method is compared to the D* Lite algorithm and results are presented. In the last section, the parameters of the robot are optimized using biogeography-based optimization (BBO). This is an efficient multivariable optimizer and it is shown that the results of optimization achieve significant improvement in robot navigation performance. In this thesis, we show that using evolutionary optimization methods like BBO can reduce the risk of collision and the navigation time by about 25% each. The resulting risk of collision indicates safe navigation by the robot which leads to the conclusion that this is a feasible method for real-world robots.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Mobile robots have been become increasingly popular in automated industrial environments. Surveillance, Mars explorers, and underwater exploration are other applications of mobile robots. In many situations, the robot doesn't have prior knowledge about the environment and providing this information is either difficult or impossible. In all of these applications collision-free path planning is an important necessity. Furthermore, using a robot in any of the aforementioned situations is acceptable only if the robot provides high efficiency and low cost. High efficiency can defined as fast customer service, low power consumption and low maintenance costs. Therefore, the robot should have the ability to autonomously find a path with a minimum risk of collision and high efficiency with no knowledge, or with minimum knowledge, about the surrounding environment. This chapter reviews previous work and research that has been done to make robot navigation safe and efficient.

Path planning algorithms are also used in applications other than robot navigation. In [25], the authors used path planning algorithms to find hierarchical routes for networks in wireless mobile communication. The authors in [6] used sampling-based path planning algorithms to define flexible molecular models as conformational filters with energy refinement to provide a geometric interpretation of constraints affecting molecular motion. They claimed that they developed new techniques for better exploitation of the geometric path information provided in the first filtering

stage. Kinematic chains are introduced in [23] as a ubiquitous representation of biological macro-molecule motion. In both robotic and biological applications of path planning, methods can benefit from a collision detection system. In [23] the authors used various benchmarks in their research and claim to have found a novel kinematic chain representation for fast detection of self-collision. Furthermore, they concluded that their approach can be used in both robotics and molecular biology. Collision detection is also important in computer animation. When several graphical objects move on a computer screen, they may interfere with each other and need to react appropriately. This is especially important if we want to have realistic collision behavior. Detecting collisions accurately is a time consuming process so designing a dynamic simulation system is required to handle this issue [26].

The motion planning problem involves searching in the configuration space of a complex geometric rigid body that connects a starting point to a destination point through a collision free path [22]. This path should satisfy the constraints imposed by other rigid bodies or obstacles in the environment. There are complete methods to solve general problems [30, 3] but they are known for their computational complexity and computing demands. So this limits the possibly of using them for even low-dimensional configuration spaces. In [22], the authors conclude that a randomized approach called rapidly-exploring random trees (RRTs) in single-query motion planning yields good performance for a wide range of problems and applications. They claim that their method is probabilistically complete and is suitable for incremental distance computation algorithms. In [22] the authors improve the robot's behavior by optimizing the RRT step sizes.

Service robots in uncertain environments have become very popular in the last few years. A variety of systems exist in places like hospitals, office buildings, department stores, and museums. As mentioned earlier, for robots to be capable team members and helpful assistants, especially in dynamic human-populated envi-

ronments, they need to navigate efficiently and safely. Recall that efficiency is defined as the ability of a robot to reach a target point from a starting position in a prior unknown environment in a short time period, and safety is defined as the number of collisions a robot may have with obstacles (e.g., humans) during navigation to the target point. Robot motion planning in dynamic environments has recently received substantial attention due to the advent of autonomous cars and the growing interest in social, service, and assistive robots.

In [9], the authors focus on learning a robot path for ease of movement, and detecting and avoiding obstacles using a single camera and a laser source. In [28] the authors propose a hybrid potential field, which can be computed in real time, to navigate in a dynamic environment with 50 randomly moving obstacles. A fuzzy inference system with an accelerate / break module is developed in [35] for real-time navigation of autonomous underwater vehicles in both static and dynamic three-dimensional environments while automatically avoiding the dynamic obstacles using sonar, along with virtual acceleration and velocity in both the horizontal and vertical plane. In [17], a fuzzy truck control system for obstacle avoidance with reasonably good trajectory is proposed, using 33 fuzzy inference rules for steering control and 13 rules for speed control. In [13], two fuzzy logic controllers for steering and velocity control of an autonomous vehicle are divided into seven control modules. This leads to the generation of a path to the target point, including desired orientation, while avoiding collisions with obstacles, driving the vehicle through mazes, and controlling the velocity based on the obstacles in the map and based on the need to navigate around sharp corners.

A novel algorithm for collision free navigation in complex dynamic environments with moving obstacles is proposed in [29]. The algorithm considers an integrated representation of the environment by approximating the shape of the obstacles in the map by circles and polygons. This reduces the required computational

effort and increases the speed of the simulations. The navigation algorithm provides minimum-distance path planning through a crowd of moving or stationary obstacles [29].

An efficient stereovision-based motion compensation method for moving robots is presented in [15] using the disparity map and three modules: segmentation, feature extraction, and estimation. In the segmentation module, the authors propose the use of extended type-2 fuzzy information theory to recognize the obstacles. Fuzzy logic is used to implement the design and coordination [36] of a memory grid and to develop a minimum risk method for robot navigation, and is able to avoid collision with obstacles in different scenarios, such as long walls, large concave and recursive UU-shaped regions, unstructured regions, cluttered regions, and maze-like obstacles that represent dynamic indoor environments. In [1] a fuzzy logic controller is developed based on the Mamdani-type fuzzy method for robot navigation and obstacle avoidance in a cluttered environment. A fuzzy controller with three inputs and a single output provides safe navigation for the robot motion in a static environment while taking into account the accuracy of the measurements of its position, distance to the obstacles and the goal point, speed, orientation, and the rate of change of its heading angle. The authors in [1] describe a fast and reliable method of obstacle avoidance for both for outdoor and indoor navigation. The method is applicable in various mobile robotic systems regardless of whic sensors are used and is based on two complementary approaches: non-complex implementation and human-like smooth steering. In [37] a conceptual approach is considered based on fuzzy logic to solve the local navigation and obstacle avoidance problem for multi-link robots. The fuzzy rule-based approach is considered as an on-line local navigation method for the generation of instantaneous collision-free trajectories.

The above papers use fuzzy approaches, but there has also been research with probabilistic approaches. In [16], the authors proposed a method in which a multi-

degree of freedom robot uses two-step path planning: the first step uses a probabilistic method to generate all possible paths between its configurations, and the next step, the query phase, connects the generated paths between two nodes and selects the best path. This method was experimentally demonstrated with pre-known static maps. Rapidly-exploring random trees(RRTs) are introduced in [22]. RRTs incrementally generate new nodes from the starting node to the ending node. The nodes explore the map using simple greedy heuristics.

Path planning algorithms can be optimized using different algorithms. In general, optimization means selecting the best variables relative to some criterion from a set of available options. Selecting the best variables will cause a function called the cost function to be maximized or minimized.

Many approaches have been developed to optimize motion planning algorithms. Common motion planning algorithms produce inefficient trajectories for high dimensional and complex dynamics [8, 18, 5]. There are various methods for optimization; genetic algorithms, particle swarm optimization (PSO) and biogeography-based optimization (BBO) are a few of them. In this thesis, BBO is used as the optimization method. PSO individuals tend to clump together, but BBO doesn't have that limitation [31].

There are uncertainties in all evolutionary algorithms. The first type of uncertainty is caused by the noise of the cost function evalution, which can have many different sources, such as noise from measurement sensors. The second type of uncertainty is caused by the nondeterministic behavior of the environment. This means that environmental parameters change with time and optimization should be robust enough to adapt to new parameters. The environment can vary after the optimum values are found, but the optimum parameter values should still give satisfactory results. The third type of uncertainty comes from the approximation of the cost function. The cost function is estimated by the most feasible and appropriate form

(meta-model) due to the complexity and difficulty of using the actual cost function. This will add errors and uncertainties. For the last type of uncertainty, systems need to be continuously optimized. The optimizer should be able to track the optimum parameter values over time. The challenge here is to use previously generated optimum values to speed up the new optimization process [14].

Contribution of this Research

In this research, a probabilistic method is introduced for path planning. This method is fast, real-time and can be extended to various types of autonomous robots. It can be used for autonomous underwater exploration and the same algorithm can used for autonomous drones. The robot uses a minimal amount of memory.

To optimize the path planning algorithm, the parameters of the method are tuned using biogeography-based optimization (BBO). The BBO method in this research optimizes 14 parameters for the path planning algorithm.

CHAPTER II

PROBABILISTIC PATH PLANNING

This chapter describes our new probabilistic path planning algorithm. The path planning problem is set in a two dimensional map, in which the robot is trying to move from a starting point to an arbitrary but known target point. The robot does not have prior knowledge about the shape, location and size of the obstacles in the map and so it begins by hypothesizing that the path to the target point is a straight line from its current location.

The map is treated as a continuous environment and the location of the obstacles and the robot is determined in (x, y) coordinates. The map contains dynamic and static obstacles, an example of which is shown in Figure 1. The robot is equipped with a 360° range-limited radar sensor that is used to find the distance to the obstacles around the robot. The radar system is explained in detail in the next section. The robot also has the ability to instantaneously change its velocity to avoid collisions.

The objective of the robot is to find a minimum-time trajectory starting from the initial point on the map to the target point while minimizing the probability of collision with the static and dynamic obstacles.

We assume for now that the robot has a perfect sense of its current location as well as the location of the target point. The robot will determine the heading angle α with which to reach the target point if there were no obstacles in the way. However, there are might be unknown obstacles on the way and the robot needs

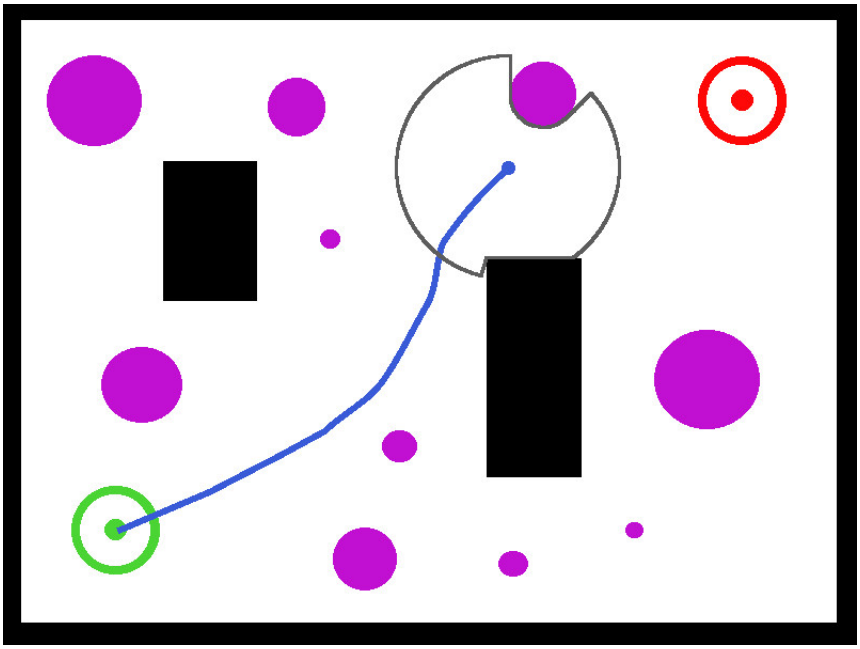


Figure 1: Sample map with static obstacles in black and dynamic obstacles in pink. Dark gray shows the radar range, the red symbol in the upper right shows the target point and the blue line shows the path of the robot starting from the initial point at the lower left, which is marked in green.

to react accordingly. So, in our algorithm, the robot creates a normal Gaussian distribution d_T centered at α with a dynamic adjustable standard deviation σ. The robot uses a function f_θ to return the argument of the maximum value of an input distribution as the best angle to travel to avoid colliding with obstacles. The output of f_θ with d_T as its input is clearly α.

$$f_\theta(d_T) = \text{argmax}(d_T) = \alpha \tag{2.1}$$

The probabilistic algorithm dictates that the robot choose an angle Θ to direct the robot to the target point in a way that has the lowest probability of colliding with an obstacle, or in other words, has the highest probability of reaching the destination without a collision. To find Θ, we construct three distributions; one is d_T as mentioned above, which we call the target distribution. The target distribution is created on the basis of the current location of the robot and the target point is Gaussian with center

α. The second distribution is called the obstacle distribution, d_O, and is created based on the obstacles detected by the robot's radar. The third distribution is called the memory distribution, d_M, and uses locations that have been previously visited by the robot to create a Gaussian distribution. This distribution helps the robot escape rooms or blocked areas by backtracking.

2.1 Radar System

The robot does not know anything about its environment except the information that it obtains from its radar. The robot is equipped with a 360° radar that it uses to obtain information about its surrounding area, including both static and dynamic obstacles. Each time step, the radar rotates once and collects 360 data points, one data point per degree. That is, the radar is configured with a 1° resolution.

This arrangement is shown in Figure 2. The robot coordinates are denoted as $[X_R, Y_R]$. In practice, the laser beam emits a ray and as the ray bounces from an obstacle, the sensor measures the distance based on the elapsed time. However, in simulation, the distance of the robot to an obstacle is calculated with geometric equations.

Although the robot has no prior information about the obstacles or the environment, the simulator has complete information for all objects in the simulation. So the robot emits the laser beam and then the simulation program provides the distance to each obstacle. This approach makes it possible to have modular code that can be easily ported and converted to a real-world application. Because the path planning algorithm does not need to know anything about how its radar works, the robot can simply request the distance to the closest obstacle at a specific angle and then wait for the answer from either the simulator or the radar module. Therefore, the obstacle distance can be generated using either the simulator or a real-world laser beam sensor.

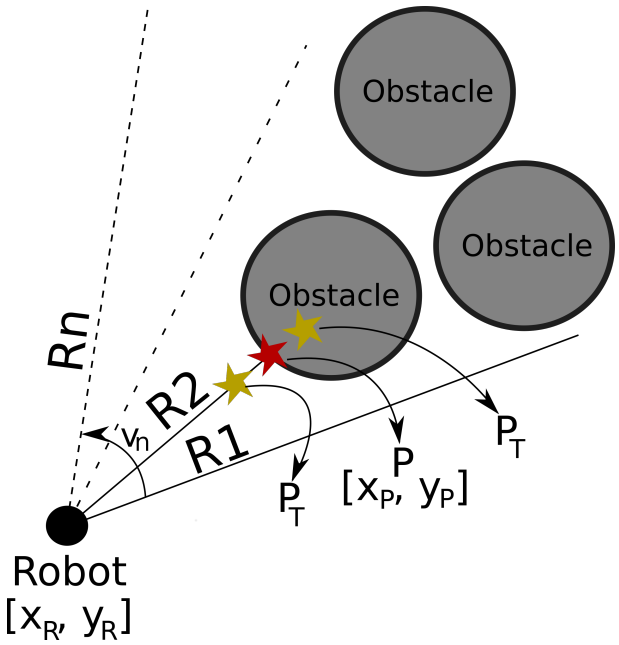


Figure 2: Laser beam rays are denoted as R_1, R_2, $\cdots$, R_n. The approximate intersection points of the radar with the obstacle are denoted as P_T, which, because of the radar resolution, is not the same as the exact intersection point P. ν_n is the angle of laser beam n with the horizontal axis.

In our simulations, three methods were simulated to calculate the distances from the robot to the obstacles. In the first method the map is digitized to create a grid-based environment. The second method considers a continuous-space (infinite resolution) map and characterizes the objects as simple mathematical shapes. Then, using mathematical formulas, the intersection of the beam and each object is computed. In the third method, to increase the simulation speed, a physics engine library is used, which computes the intersection of the beam ray and the objects using optimized ray-casting algorithms. Each of these methods are explained in more detail in the following.

In Figure 2 the location of each circular obstacle and its radius is known. To calculate the intersection of the laser beams (denoted as $\{R_1, R_2, \cdots, R_n\}$ at angles $\{\nu_1, \nu_2, \cdots, \nu_n\}$) and the circular obstacle, the simulator creates a right triangle as in Figure 3. The sides of this triangle are shown as $Y_{O,j}, X_{O,j}, S_{O,j}$, where $j \in \{0, 1, ..., n\}$ and n is the number of steps and is defined experimentally. $\{S_{O,j}\}$ denotes the steps

that the simulator uses to see if the radar beam is intersecting with free space or an obstacle. n in figure 3 equals to 6. Line segments start from the robot and the following equation is used to find the coordinate of each radar step:

$$\begin{aligned} Y_{o,j} &= \sin(\nu)\xi\frac{j}{n} + y_R \\ X_{o,j} &= \cos(\nu)\xi\frac{j}{n} + x_R \end{aligned} \tag{2.2}$$

for $j = 0, 1, \cdots, n$, where ξ is the maximum range of the radar beam, which in our experiments equals 10 meters. It also should be noted that in the Equation 2.2, the maximum value of $\frac{j}{n}$ is 1. If $S_{O,j}$ is located on an obstacle, the simulator returns $\xi \times \frac{j}{n}$ as the distance returned by the laser beam ray. This method requires the map to be implemented as a grid-based environment. Low resolutions will result in faster computation time but the movement of the robot and the obstacles may not be smooth, so the algorithm may not be extendable to real-world applications. Higher resolution maps would result in higher memory usage and slower computation speed. A map with the size 800×600 pixels and a resolution of 1 step per pixel would take about 20 seconds to be processed. Furthermore, grid-based maps are prone to errors, as shown in Figure 2, where yellow marks denoted as P_T indicate the locations that this method can report, which are not accurate since their coordinates are estimated only to within a given resolution. The need to implement a map with both near-perfect resolution and fast simulation speed was the motivation for a new solution and method, as explained below.

To achieve perfect radar range resolution, the grid approach was removed from the simulation. Rather than going through steps in Figure 3, mathematical equations were used to compute distance. In order to accomplish this, the obstacles were categorized into one of two types: circular obstacles, and obstacles composed of straight lines. These cases are discussed in the following sections.

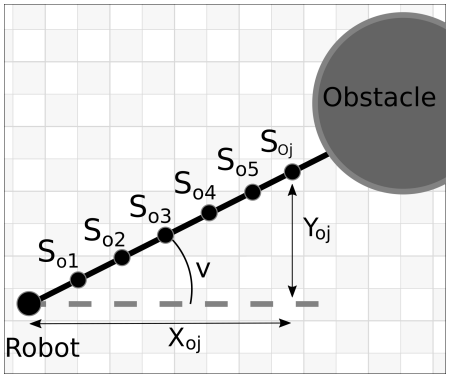


Figure 3: A sample ray of the laser beam and the steps that the simulator uses to check whether each point along the beam is free space or occupied by an obstacle.

2.1.1 Circular Obstacles

In order to show a circular obstacle on a map, we need the coordinates of the center and the radius of the obstacle. For a circular obstacle, these two values are stored in a variable in memory. The center of the obstacle is denoted as $C\{x, y\}$ and the radius is denoted as C_R. The location of the robot in Figure 4 is denoted as $[x_R, y_R]$ and is known to both the robot and the simulator. To calculate the distance from the robot to the obstacle, we first find the line equation of the laser beam in the form of $y = mx + n$. Denoting the angle of the beam as ν, we have the equations

$$\begin{aligned} m &= \tan(\nu) \\ n &= y_R - mx_R \\ y &= mx + n \end{aligned} \tag{2.3}$$

where y_R and x_R are the coordinates of the robot. We also need to know the extent of the laser beam, which location we denote as Q and whose coordinates we denote as $[X_Q, Y_Q]$, as shown in Figure 4. The following equation shows how to compute the coordinates of Q.

$$\begin{aligned} x_Q &= \cos(\nu)\xi + x_R \\ y_Q &= \sin(\nu)\xi + y_R \end{aligned} \tag{2.4}$$

where ξ is the radar radius.

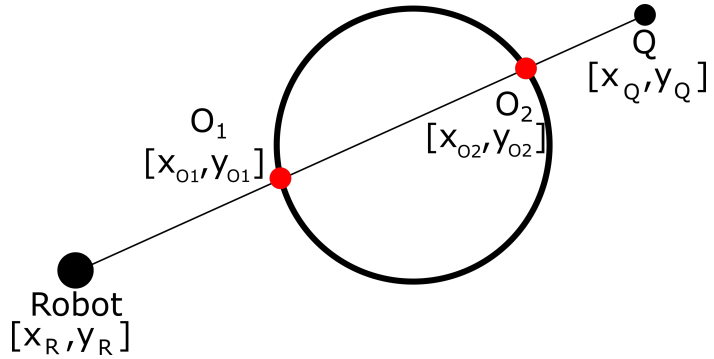


Figure 4: Intersection of a line segment, which denotes the robot radar beam, and a circular obstacle.

For convenience we denote all points relative to the center of the obstacle. We can find the intersections using the following equations:

$$\begin{aligned} d_X &= x_Q - x_R \\ d_Y &= y_Q - y_R \\ D &= \sqrt{d_X^2 + d_Y^2} \\ D_t &= \begin{vmatrix} x_R & x_Q \\ y_R & y_Q \end{vmatrix} \\ \Delta &= R^2 D^2 - D_t \\ [O_{X1,X2}] &= \frac{D_t d_Y \pm d_X \sqrt{\Delta}}{D^2} \\ [O_{Y1,Y2}] &= -\frac{D_t d_X \pm d_Y \sqrt{\Delta}}{D^2} \end{aligned} \tag{2.5}$$

where R is the radius of the obstacle. If Δ is negative there is no intersection, if $\Delta = 0$ there is one intersection, and if $\Delta \geq 0$ there are two intersections. If there is more than one intersection, O_1 and O_2 in Figure 4, the intersection point that is closer to the robot provides the desired distance measurement.

2.1.2 Obstacles composed of straight lines

The other obstacles in the simulator are obstacles with straight lines, such as rectangles. Figure 5 illustrates this type of obstacle. In Figure 5, $C1$, $C2$, $C3$ and $C4$ are the coordinates of the corners of the rectangular obstacle. To find the coordinates of the intersections of the radar beam with the rectangle, which are shown as Z_1 and Z_2, the simulator compares each line segment of the obstacle with the laser beam ray and then saves those lines that intersect with the laser beam. In the next step, the simulator finds the distance from the robot to the intersections and selects the intersection point that is closest to the robot as the robot-obstacle distance. Here is an example for Figure 5.

1. Does line segment $C1 - C2$ intersect with R? **True**
2. Does line segment $C2 - C3$ intersect with R? **False**
3. Does line segment $C3 - C4$ intersect with R? **False**
4. Does line segment $C4 - C1$ intersect with R? **True**
5. Find the intersection of $C1 - C2$ and R and call it Z_2
6. Find the intersection of $C4 - C1$ and R and call it Z_1
7. Return min(distance(Robot, Z_1), distance(Robot, Z_2))

The aforementioned types of obstacles can be combined to make different obstacles with various shapes. Using the methods above, the resolution of the radar is mathematically perfect and doesn't depend on the resolution of the map. Other advantages of this method are that it is faster than grid-based maps and requires very small memory since the simulator needs to store just a few properties of the obstacles and not all of their coordinates. However, this method could be further improved by limiting the intersection calculations so that the simulator doesn't compute the

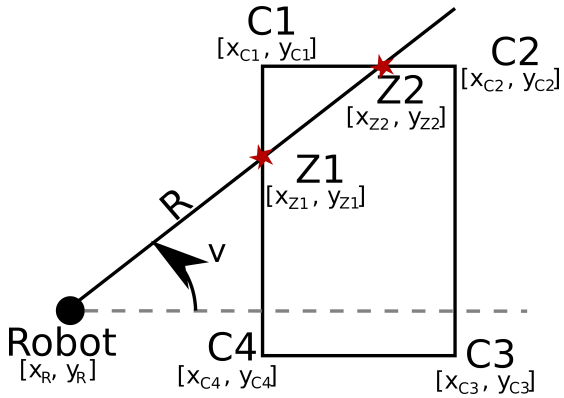


Figure 5: The corners of a rectangular obstacle are denoted as C_1, C_2, C_3 and C_4. The positions of the corners are know to the simulator. The intersections of the beam ray R and the obstacle are denoted as Z_1 and Z_2.

intersection of the ray with all obstacles in the map. Nevertheless, by using a non-optimized version of the algorithm the simulation speed was improved from 20 seconds with the grid-based approach to approximately 8 seconds.

Having optimizing the simulation, different physics engine libraries were used in this research. The Box2D [4] physics engine was used for the ray-casting of the radar system and for handling collisions. Box2D uses an optimized code to create a virtual world and simulate the physics of the objects in the world. So the obstacles were converted from mathematically defined objects to objects in the format required by Box2D. Box2D also uses metric units which makes the behavior of the autonomous agent more natural and portable to real-world applications. The radius of the radar in our simulation is 10 meters. Also, the use of this library significantly improved the simulation time from 8 seconds to about 3 seconds.

To conclude, the radar system calculates the distance of the nearest obstacle in 1° increments and stores the distances in an array. The array looks like Figure 6. Distance is normalized to the range $[0, 1]$. This means that if there is no obstacle in a certain direction, the distance is 1, and if an obstacle is touching the robot, the distance is 0. The memory array has 360 blocks for each step of the radar laser beam. This number can be increased or decreased. However, the value of 360 was

chosen because it returns a result with appropriate resolution without increasing the simulation time unnecessarily. Increasing the number of blocks, or decreasing the step size to a value less than 1°, would help the robot detect smaller obstacles. In our simulations, the obstacles' size is not tiny so our current radar resolution can be used without being worried about missing an obstacles with the radar.

index	distance	index	distance	index	distance
0	1	9	1	351	0.03
1	1	10	1	352	0.01
2	1	11	1	353	0.13
3	0.6	.	.	354	0.12
4	0.5	.	.	355	0.5
5	0.3	.	.	356	0.12
6	0.5	.	.	357	0.15
7	0.6	.	.	358	1
8	1	.	.	359	1

Figure 6: Sample memory array returned by the radar system. This memory array contains 360 entries, one for each degree of radar resolution.

2.2 Target Distribution d_T

To construct d_T, the robot ignores all obstacles. The distribution domain is $[0, 2\pi)$ and is defined as

$$d_T(\phi) = \frac{A}{\sqrt{2\pi\sigma_T^2}} e^{-\frac{(\phi-\alpha)^2}{2\sigma_T^2}} \qquad \phi \in [0, 2\pi) \tag{2.6}$$

where A is the distribution's dynamic amplitude coefficient; in our experiments, $A = 1$. α is the angle toward the target and is computed by the robot, and σ_T is the standard deviation and is a user-selectable parameter. Figure 7 shows two target distributions with two different values for σ.

Changing σ_T will impact the robot's behavior. Smaller values will force the robot to choose a path close to the obstacles, and higher values will give the robot

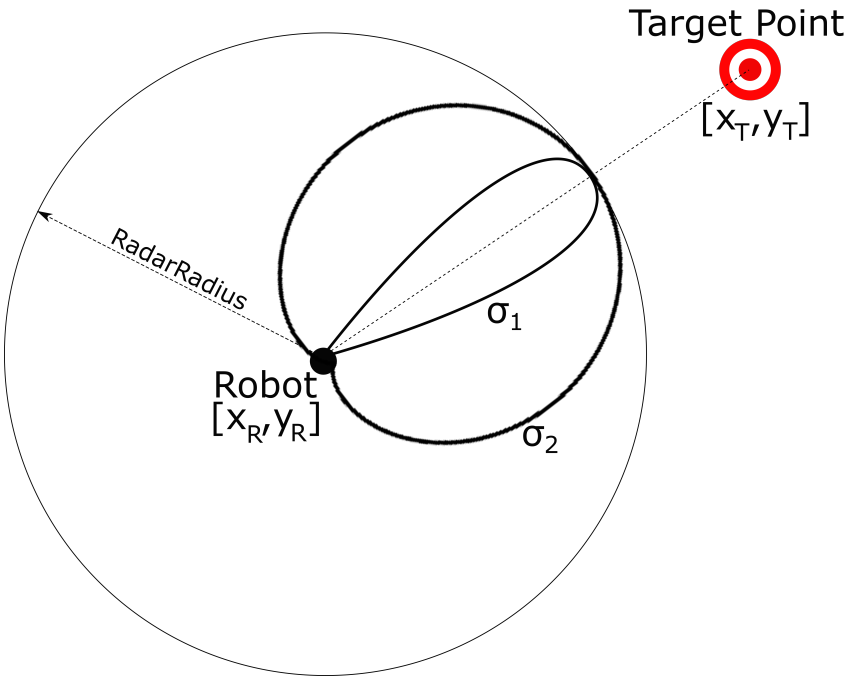


Figure 7: Two target distributions with different σ values.

a greater higher turning radius around the obstacles, resulting in a safer trajectory with less risk of collision. However, a greater avoidance of the obstacles will result in a greater travel distance to the target, which will result in an increase in traveling time. BBO will be used to optimize the value for σ_T, as explained in the next chapter.

2.3 Obstacle Distribution d_O

The obstacle distribution is constructed by the 360° radar sensor on the robot. The radar system was explained in section 2.1. The obstacle distribution d_O is a function of the distance sensed by the radar to the nearest obstacle at each angular position around the robot. The output of f_θ in Equation 2.1 for input d_O will return an angle at which the robot has the lowest probability of colliding with an obstacle. In Section 2.1 it was explained that the radar distance array contains normalized values

for distance. We define the following equation for d'_O, which normalizes d_O:

$$d'_O = \frac{d_O}{\gamma} \tag{2.7}$$

where γ is a coefficient that affects how sensitive the robot is to obstacles. If γ is large the robot will be less sensitive. In our experiments, the value of γ is the maximum range of the radar, which results in

$$\max(d'_O(\phi)) = 1, \quad \phi \in [0, 2\pi) \tag{2.8}$$

The minimum value of d_O is zero, which occurs when an obstacle is touching the robot. In our experiments this happens rarely since the robot tends to move away from obstacles. In our experiments, this happens only if the speed of an obstacle is higher than that of the robot. In the following sections, the speed of the robot is discussed in more detail. There are some situations where the maximum speed of the robot is less than the speed of the obstacles, which could easily result in a collision. Figure 8 shows a sample obstacle distribution (d') with maximum distance 1, which means there are no obstacles in that direction; these directions are indicated in the figure with the symbol B. If the distribution value is less than 1 in some direction, indicated with an A in the figure, an obstacle is detected within the radar range in that direction.

The simplest situation in simulation considers the robot as a point with zero radius. However, in the real world, the robot has a nonzero dimension. In order to establish a safe margin when the robot tries to go around the corner of an obstacle, we define a threshold J_{dF} which shifts the obstacle distribution down. This means the robot treats the obstacles as closer than their real positions. The value of this parameter is a function of the robot dimensions but is determined experimentally and is optimized later in this thesis.

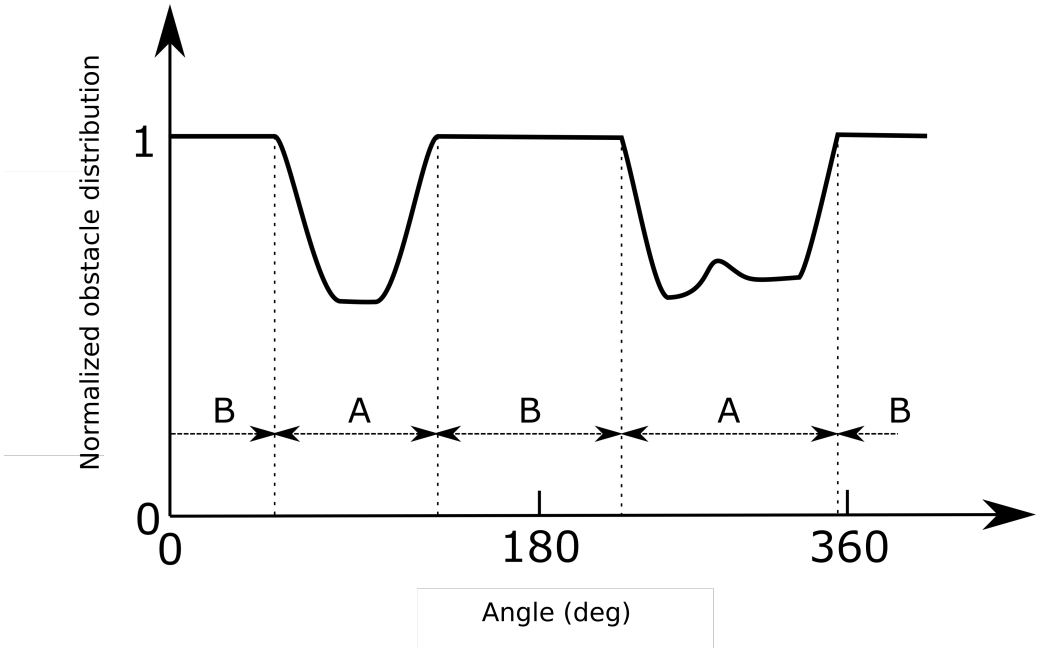


Figure 8: Obstacle distribution d'. There are two types of regions in this figure: A indicates the existence of obstacles at the given angles, and B indicates that there are no obstacles at the given angles.

2.4 Final target distribution d_F

The algorithm uses d_T and d'_O to construct the final target distribution d_F, which tells the robot which direction to choose to prevent colliding with obstacles while still moving as directly as possible toward the target. The algorithm constructs d_F by taking the minimum value of d_T and d'_0 at each angle $\phi \in [0, 2\pi)$:

$$d_F(\phi) = \min(d_T, d'_O), \quad \phi \in [0, 2\pi) \tag{2.9}$$

By using the minimum operation to construct the final target distribution, directions toward obstacles have a lower distribution value, and directions away from obstacles have a higher distribution value. The f_θ function, described earlier, can be used to find Θ, which is the argument of the maximum value of d_F, which will be the most favorable angle for a trajectory that is both safe and in the direction of the target point. Figure 9 shows an example of d_F.

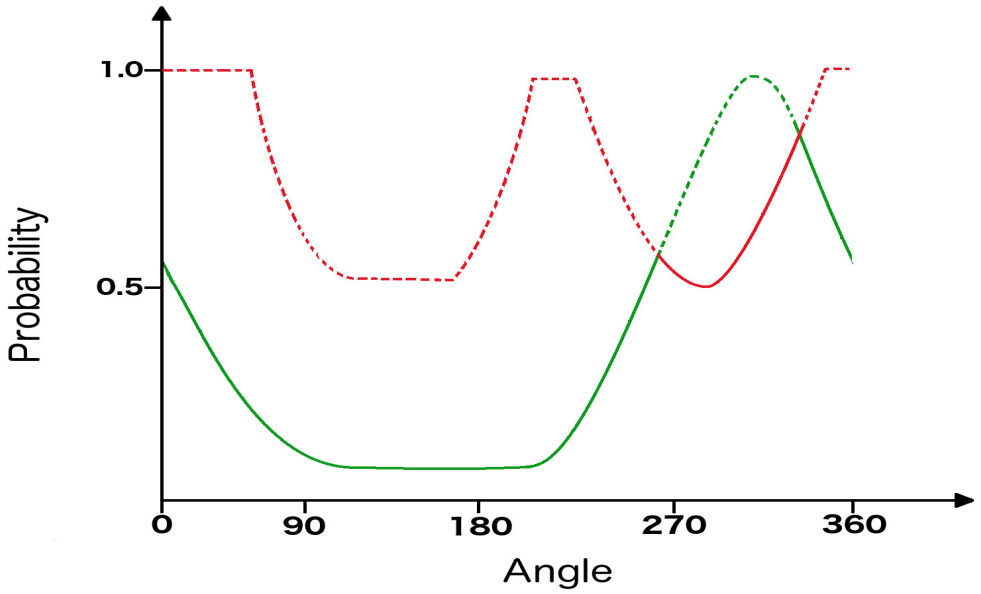


Figure 9: Example of final target distribution construction. The original target distribution is green and the obstacle distribution is red. The solid line is the final target distribution and is constructed by taking the minimum value of the original target distribution and the obstacle distribution.

The robot also changes its speed depending on the location of the closest obstacle. Figure 10 depicts the speed algorithm of the robot. The robot has five speed modes: very slow ($S1$), slow ($S2$), normal ($S3$), fast($S4$), and very fast ($S5$). The velocity of each speed mode will be optimized by the BBO algorithm. The robot changes its speed based on the location of the nearest obstacle according to Figure 10.

In Figure 10, the robot is shown as a black filled circle in the middle of the figure, and the direction of motion is toward the top of the page and is shown with an arrow. The radius of the largest circle is L, which is the maximum range of the 360° radar. As the robot navigates and passes obstacles, some obstacles might enter the regions A, B, C, or D. If the closest obstacle is in region A, the robot will choose speed mode $S2$. Region B is associated with speed mode $S1$, and regions C, D, E, and F are associated with speed modes $S5$, $S4$, $S3$, and $S3$ respectively. If there is an obstacle in region E or F the robot will ignore them and will continue with normal speed $S3$. Region B is a symmetric arc with angle $2\theta_2$ and is taken from a circle

centered on the robot with radius r_B. Likewise, region C is a symmetric arc with angle $360-2\theta_1$ and is taken from a circle centered on the robot with radius r_C. r_B and r_C are independent and belong to $[0, L]$. Parameters θ_1, θ_2, r_B and r_C are empirically determined and highly affect the safety of the robot. They will be optimized later in this thesis with the BBO algorithm. The parameters are summarized in Table I.

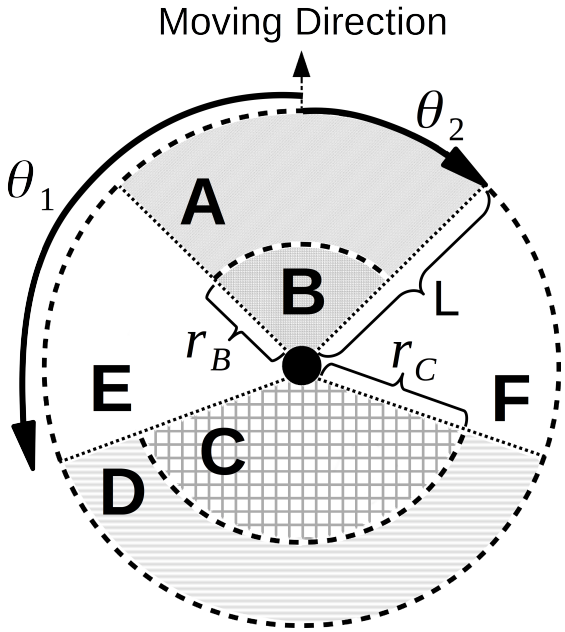


Figure 10: Different speed mode regions that the robot uses to change its speed according to the position of the closest obstacle.

Location of closest obstacle	Radius	Speed mode
Region A	L (radar range)	$S2$ (Slow)
Region B	r_B	$S1$ (Very Slow)
Region C	r_C	$S5$ (Very Fast)
Region D	L (radar range)	$S4$ (Fast)
Region E	L (radar range)	$S3$ (Normal)
Region F	L (radar range)	$S3$ (Normal)

Table I: Speed modes; see Figure 10 for regions.

2.5 Memory Distribution d_M

The final target distribution d_F constructed in the previous section is highly goal-oriented. In complex maps, like mazes or buildings with multiple rooms, the robot might not be able to reach the target without backtracking (that is, returning to regions or rooms that it has already visited). However, current global robot path planners require the robot to have a memory to store previously visited locations on the map. Building the map or reconstructing the map can also be implemented with our algorithm, but this is not our goal since we want to develop an algorithm that uses as little memory as possible. Our algorithm's memory is a queue that can save up to 500 previously visited coordinates. When the number of time steps exceeds 500, the earliest elements are discarded. The robot uses this queue to escape rooms or corners in which it might become stuck. This allows the robot to escape rooms and to backtrack through previously visited regions. Figure 11 depicts the memory queue. Note that the number of memory blocks in the queue can be changed and can be optimized. However, in our simulations, the number of the blocks is constant and is equal to 500.

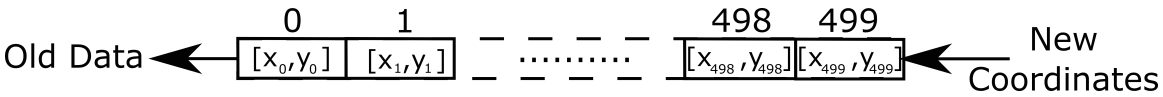


Figure 11: Memory queue in which previously visited coordinates are stored.

The robot saves the previously visited coordinates in memory as it navigates. These saved coordinates create a repulsive force. If the robot gets trapped in a room or a dead end, the memory points that are saved in that area become more dense. This will cause the robot to backtrack and escape from that area. The direction of the repulsive force follows a Gaussian distribution centered at the mean of directions of the memory points with standard deviation σ_M, which will be optimized later in this thesis with BBO.

Figure 12 shows an example of the robot and the memory points (MPs) stored in the queue (fewer than 500 for the sake of illustration). Figure 13 shows a Gaussian distribution based on the MPs. As shown in Figure 13, the MP in the middle has the highest weight.

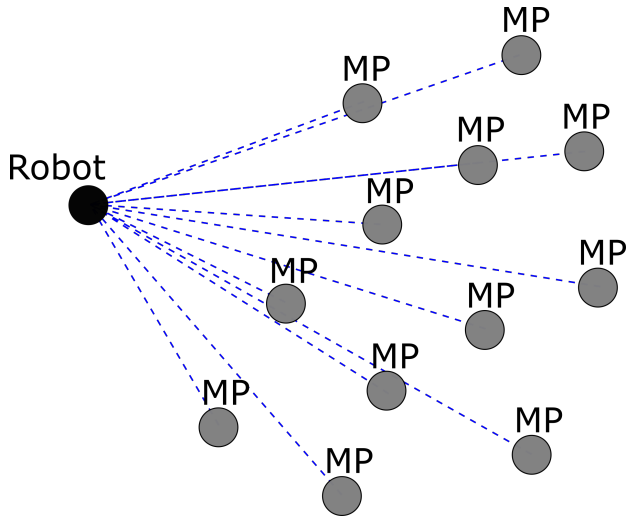


Figure 12: Illustration of memory points (MPs). The dashed lines show the direction of the repulsive force imparted to the robot by each MP.

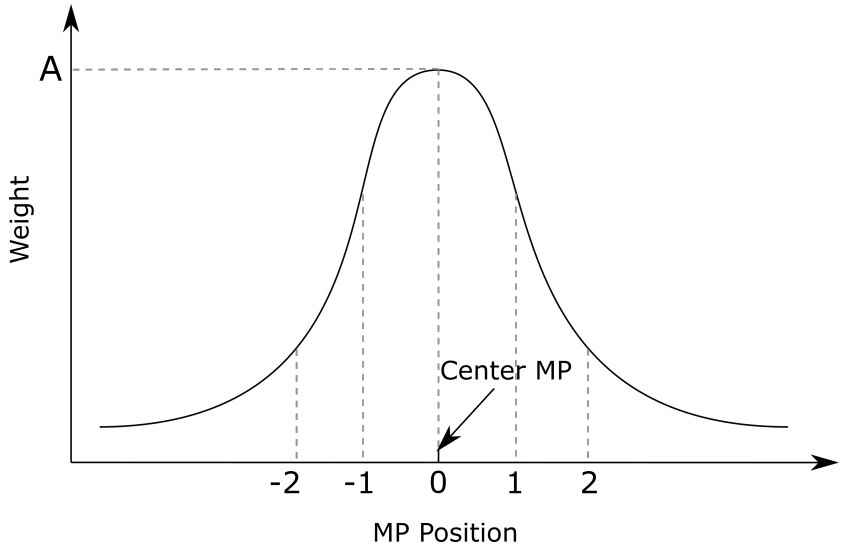


Figure 13: Memory distribution based on recently visited locations.

The navigation algorithm executes at 60 Hertz, so the 500-point memory queue contains data for the previous 8 seconds, approximately. The distance traveled in 8

seconds depends on the speed of the robot. In a real-world application, 8 seconds might not be long enough for the memory queue and the size of the queue might need to be adjusted. Each memory block will occupy 8 bytes. As mentioned above, the robot uses 500 blocks for the queue, so the total memory requirement is 8 bytes $\times$ 500 = 4000 bytes. This amount of memory is available in almost any embedded system. Table II compares the memory size required to store the coordinates of recently visited locations. This table shows that the required memory is appropriate even if we want to save 30 minutes worth of previously visited locations. However, the goal of this method is to keep the memory requirements as small as possible so that the algorithm is suitable even for small embedded systems.

Time	Size
8 sec	3.75 KBytes
16 sec	7.5 KBytes
64 sec	30 KBytes
10 min	281.25 KBytes
30 min	843.75 KBytes
1 hour	1687.5 KBytes

Table II: Memory size comparison for various lengths of time for which the robot will save previously visited points. The nominal value in this thesis is 8 seconds.

2.6 Final Robot Steering Direction: Combining d_M and d_F

At this point we have steering angles from the memory distribution and the final target distribution. We call these steering angles V_{dF} and V_{dM} respectively. In order to determine the final robot steering direction, the robot uses the memory vector only if necessary, so the memory vector has a lower weight than the final target vector on the final robot steering direction. To implement this idea, we assign weights to the two steering vectors: P_{dF} for V_{dF}, and P_{dM} for V_{dM}. The appropriate values for the weights are determined empirically and in our experiments are optimized by BBO. The resultant vector is calculated by summing the two weighted V_{dF} and V_{dM}

vectors, and we call the resultant vector V_R. Figure 14 illustrates a V_R calculation. In the figure, V_{dF} shows the final target steering direction as determined by the target location and the obstacles, and V_{dM} shows the steering direction as indicated by the robot's memory to avoid previously visited coordinates.

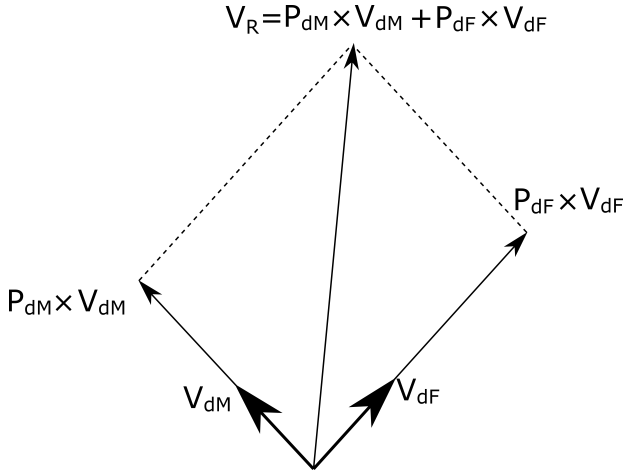


Figure 14: Illustration of d_F and d_M vectors and their weighted summation vector, which is the final robot steering direction.

Figure 15 shows an example of how the speed changes with time as the robot navigates. The speed of the robot is changes between 4 and 10 units, since most of the time obstacles are detected in front of the robot so the robot slows down to avoid collisions. For the same simulation, Figure 16 shows how the distance to the target changes with time. If there were no obstacles on the map, the distance curve would be the dashed line in the figure. However, the obstacles make the robot veer from the straight path to avoid collisions.

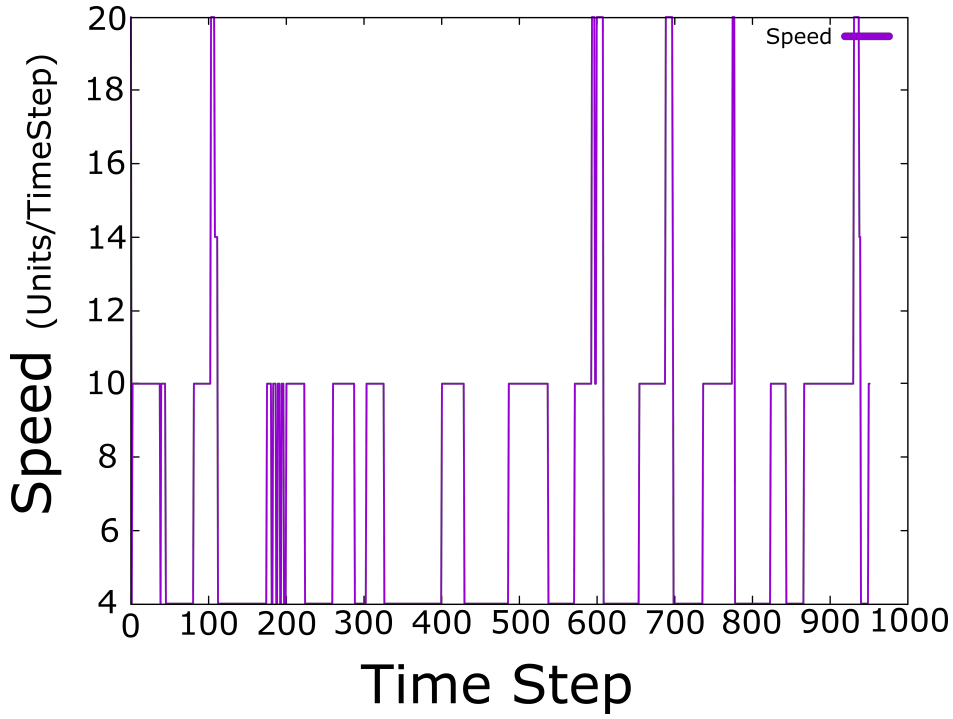


Figure 15: Illustration of how the speed of the robot changes with time based on sensed obstacles.

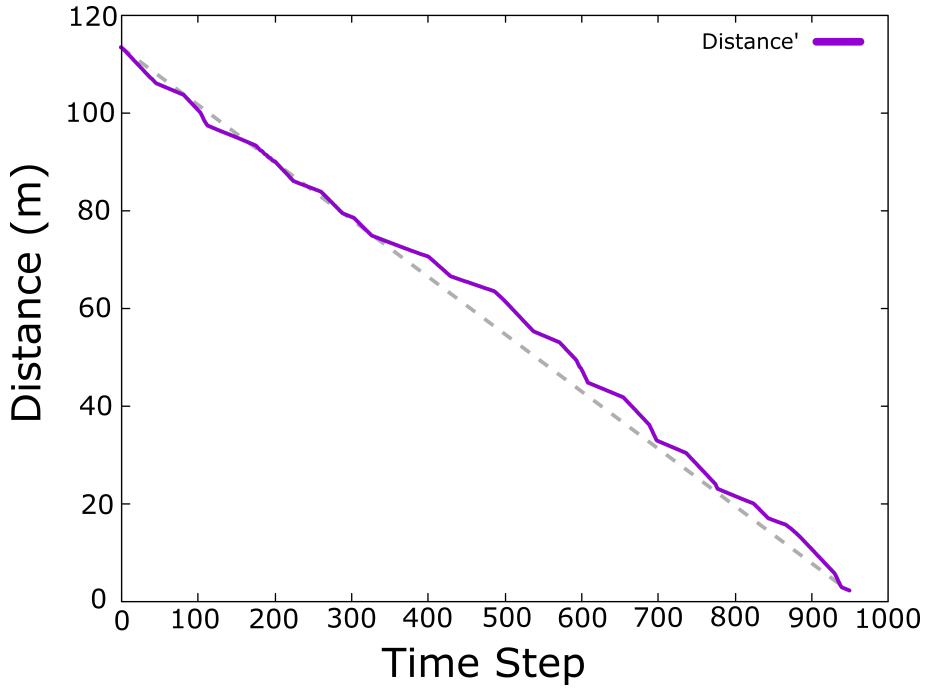


Figure 16: Illustration of how the robot moves toward the target while avoiding obstacles.

CHAPTER III

The D* ALGORITHM

The D* algorithm is based on the A* algorithm, which is a heuristic, robust, reliable path planning method that has been used in many applications [7, 10, 12]. The A* algorithm [27] searches for the best path through an environment by examining all possible paths from a known starting location to the known destination. A* considers a cost function like $f(n) = g(n) + h(n)$ where n is the current graph number, $f(n)$ is the cost to be minimized, $g(n)$ is the cost from the starting graph to graph n, and $h(n)$ is the heuristic that estimates the cost from graph n to the destination. One of the most ubiquitous path planning methods is D* [20, 32], or dynamic A*, which is a heuristic path planning method for dynamic environments. The D* algorithm is designed to plan the optimum trajectory for a robot as the robot moves through a map in real time. In this chapter, D* will be explained and compared to our newly proposed probabilistic method from the previous chapter.

3.1 Algorithm

The D* algorithm is a generalized version of A* and can be viewed as an attempt to find a sequence of state transitions through a graph. The graph node in our experiments is the position of a robot in some environment. The optimum path is the path, among all possible paths through the graph, for which the sum of transition costs is minimal. If any path through the graph is found to be not

optimal, the path is regenerated by the path planning algorithm. The cost of the robot states in the path can be defined as any measurable value, like distance, energy, time, etc. In the D* algorithm the robot uses a sensor to obtain information about the map. The map is partially known to the robot and the robot can re-plan its trajectory as it moves through the map. To generate the initial candidate optimum path, several methods have been published, such as the initial A* path and the distance-based initial path [24, 27]. In this research the distance-based initial path was implemented [24]. There are other methods that have been designed to improve the path of a robot in a graph based map, but they become highly inefficient in high resolution maps with a large number of graph nodes [2, 38]. D* provides an improvement over the previously mentioned algorithms because it can handle large maps by updating only the parts of the map that are required for the current steering decision of the robot.

D* works by first generating a path to the target point using the A* algorithm and begins moving toward the target point using that path. Figure 17 shows a map with the robot indicated with the green R and the target point with a red T. The gray areas are obstacles that are known prior by the robot. The arrows in the map show the steering directions generated by the A* algorithm. Note that the robot can move diagonally. If the robot follows the arrows, it will lead the robot to the target point. However, the tile indicated with O means that the robot is not aware if that tile is blocked or open. Based on the direction of the arrows, the robot will go between the obstacles and will arrive at the tile indicated with O.

D* has been extensively used in real robots [34]. D* star works by creating an OPEN list that includes two type of states: RAISE and LOWER. The RAISE states mean that the path cost will be increased and the LOWER states mean that the path cost will be reduced by redirecting the arrows to a new path. RAISE states increase the cost of each state by starting from a blocked state O and then sweeping outward

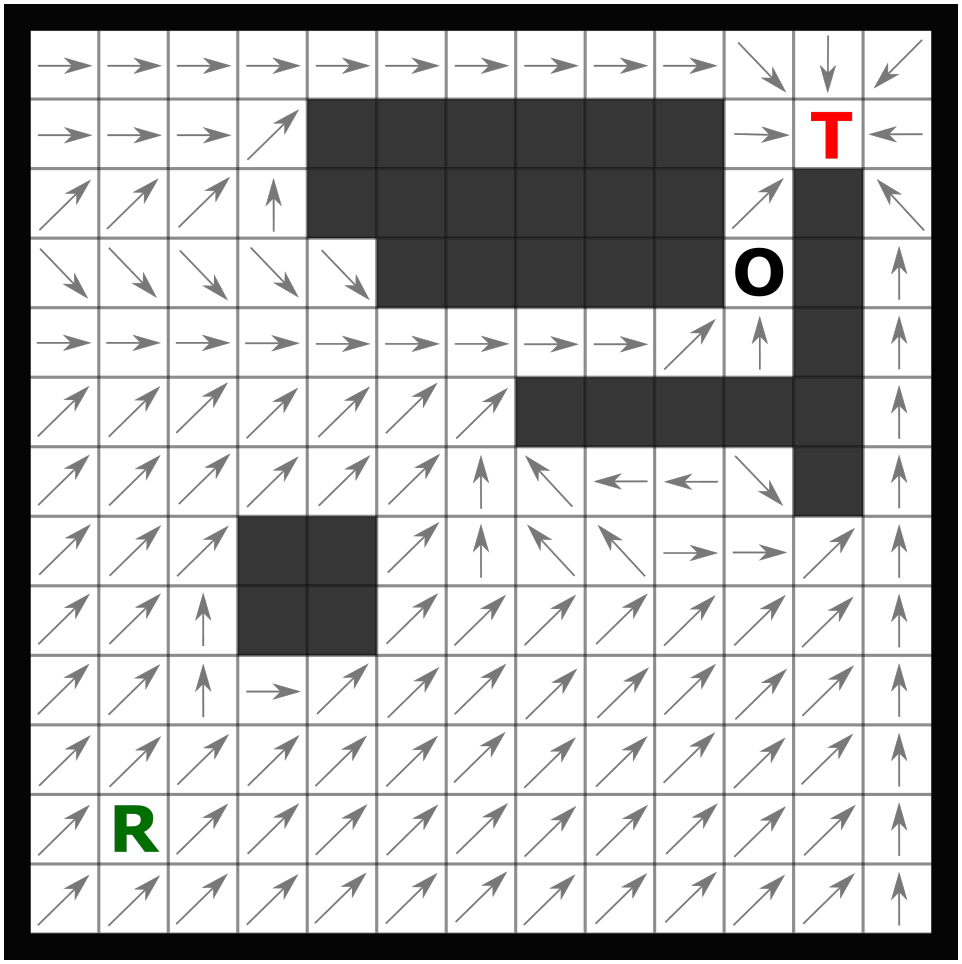


Figure 17: Sample tile map and A* navigation. The robot's initial point is indicated with an R and the target point is indicated with a T. State O shows a tile in the graph whose state (open or closed) the robot does not know. Gray static obstacles are known to the robot prior to path planning. Arrows show the optimum path to the target point as computed by the A* algorithm based on prior knowledge of the map.

to activate the neighboring LOWER states. LOWER states compute the cost and change the direction of the arrows. D*, like A* [32], can use focused heuristics to achieve cost estimation that helps the robot arrive at its destination with a lower cost, less memory, and less computing power.

The D* algorithm consists of three main functions: PROCESS-STATE, MODIFY-COST and MOVE-ROBOT. PROCESS-STATE is the first step of the algorithm and finds an preliminary path to the target point. As the robot moves through its environment,

MODIFY-COST computes new cost values for different states. If there is a state with a high cost in the pre-computed path, this function puts the state in the OPEN list and maintains the RAISE and LOWER states of all affected states. In the last step, function MOVE-ROBOT causes the robot to move in the optimally defined direction. A detailed explanation of the D* algorithm is provided in [33].

In order to enhance the D* algorithm's computational effort, it was modified [19, 21] using the lifelong planning A* algorithm to create a new algorithm called D* Lite. Although D* was implemented in this thesis, it was found to be computationally inefficient so we instead use D* Lite, which will be explained later in this chapter.

3.2 Lifelong Planning A* (LPA*)

LPA* is an enhanced and incremental version of A* and applies A* to a finite graph whose edge costs increase or decrease over time. Algorithm 1 shows the LPA* algorithm. S indicates the finite set of vertices of the graph. $Succ(s) \subset S$ indicates the successors of the current vertex ($s \in S$) of the graph. Successors are vertices that are accessible and reachable from the current vertex. $Pred(s) \subset S$ denotes the predecessors of the current vertex. Predecessors are vertices from which the current vertex ($s \in S$) can be reached. $c(s, s')$ represents the cost of moving from vertex s to vertex s', $s \in Succ(s)$. $g^*(s)$ denotes the shortest path from s_{start} to $s \in S$. LPA* always finds the shortest path between s_{start} and s_{goal} assuming knowledge of the graph topology and costs. Like A*, LPA* uses heuristics, denoted as $h(s, s_{goal})$, to approximate the distances to the goal vertex, where $s \in S$ [34]. The heuristics should obey the triangle inequality:

$$\begin{aligned} h(s_{goal}, s_{goal}) &= 0 \\ h(s, s_{goal}) &\leq c(s, s') + h(s', s_{goal}) \end{aligned} \tag{3.1}$$

Furthermore, the type of queue U will impact the behavior of the algorithm and is a priority queue. The LPA* functions are summarized in Table III.

Function	Description
U.Top()	Returns the vertex with the smallest priority in the queue U
U.TopKey()	Returns the smallest priority in the queue U
U.Pop()	Returns the smallest priority in the queue U and deletes it
U.Insert(s, k)	Inserts vertex s in queue U and sets the priority to k
U.Update(s, k)	Updates the priority of vertex s in the queue U to k
U.Remove(s)	Removes vertex s from queue U

Table III: Functions of priority vertices queue U

If queue U is EMPTY the function U.TopKey$(\cdot)$ returns $[\infty, \infty]$. The variable $rhs(s)$ in Algorithm 1 is defined as

$$rhs(s) = \begin{cases} 0, & \text{if } s = s_{start} \\ \min_{s' \in Pred(s)}(g(s') + c(s', s)), & \text{otherwise} \end{cases} \tag{3.2}$$

$rhs(s)$ is function that looks ahead to obtain better information for g-values. LPA* estimates $g(s)$ from the starting distance of the vertex s. LPA* g-values directly correspond to A* g-values.

Priorities in priority queue U are based on their key, $k(s)$. The key of each vertex consists of two components:

$$\begin{aligned} k(s) &= [k_1(s)], k_2(s)] \\ k_1(s) &= \min(g(s), rhs(s)) + h(s, s_{goal}) \\ k_2(s) &= \min(g(s), rhs(s)) \end{aligned} \tag{3.3}$$

where $k_1(s)$ corresponds to the f function of the A* algorithm $f = g + h$. LPA* g-values and rhs values correspond to A* g-values, and the LPA* h function corresponds to A* h-values. $k_2(s)$ corresponds to A* g-values. Keys are sorted by their first components, and if two keys have the same values in their first component then the

second component is compared. Vertices in priority queue U are expanded starting with the smallest key.

Algorithm 1 Lifelong Planning A*

```
1: procedure CalculateKey(s)
2:     return [min(g(s), rhs(s)) + h(s); min(g(s), rhs(s))]
3: end procedure
4: procedure Initialize
5:     U = ∅
6:     for all s ∈ S,   rhs(s) = g(s) = ∞
7:     rhs(s_start) = 0
8:     U.Insert (s_start, [h(s_start); 0])
9: end procedure
10: procedure UpdateVertex(u)
11:     if (u ≠ s_start) then
12:         rhs(u) = min_{s'∈pred(u)}(g(s') + c(s', u))
13:     end if
14:     if (u ∈ U) then
15:         U.Remove(u)
16:     end if
17:     if (g(u) ≠ rhs(u)) then
18:         U.Insert(u, CalculateKey(u))
19:     end if
20: end procedure
21: procedure ComputeShortestPath
22:     while (U.TopKey() < CalculateKey(s_goal)   OR   rhs(s_goal) ≠ g(s_goal)) do
23:         u = U.Pop()
24:         if (g(u) > rhs(u)) then
25:             g(u) = rhs(u)
26:             for all (s ∈ Succ(u)) do
27:                 UpdateVertex(s)
28:             end for
29:         else
30:             g(u) = ∞
31:             for all (s ∈ Succ(u) ⋃ U) do
32:                 UpdateVertex(s)
33:             end for
34:         end if
35:     end while
36: end procedure
37: procedure Main(())
38:     Initialize()
39:     while (True) do
40:         ComputeShortestPath()
```

```
41:         Wait for changes in edge costs
42:         for all (Directed edges (u, v) with changed edge costs do
43:             Update the edge cost c(u, v)
44:             UpdateVertex(v)
45:         end for
46:     end while
47: end procedure
```

The LPA* algorithm was implemented in C++ for this thesis. This is required to implement D* Lite, which is described in the next section and which uses LPA* as an efficient part of its path planning algorithm.

3.3 D* Lite

In the previous section we described the LPA* algorithm. LPA* uses the edge costs of the graph to find the shortest path between the start vertex and the goal vertex. The D* algorithm uses the LPA* algorithm to find the shortest path as the robot navigates through a map. The D* algorithm is summarized in Algorithm 2. D* Lite is a path planning algorithm that can be used in unknown environments. For our experiments, the initial cost of each edge is 1. If there is no clear path to the goal vertex, the cost of the edges are set to infinity. In order to implement the D* Lite algorithm, the robot assumes that current location of the robot is s_{start} and then finds the path to s_{goal}.

Note that LPA* finds a path from s_{start} to s_{goal}, so the g-values are estimates from the starting point. However, in D* Lite, the direction is reversed and the g-values are estimates from the goal point, so it finds the path from s_{goal} to s_{start}. To address this issue in our research, when the map is not known but the immediate surroundings of the robot is known, the location of the start and goal vertices is exchanged in LPA*. Figure 18 shows the updated arrows as the robot navigates through the map. D* Lite is easier to implement and debug than the D* algorithm, and is claimed to be faster as well. So in this research, the D* Lite was implemented

and is compared to our newly proposed path planning method.

Algorithm 2 D* Lite

```
1: procedure CalculateKey(s)
2:     return [min(g(s), rhs(s)) + h(s); min(g(s), rhs(s))]
3: end procedure
4: procedure Initialize
5:     U = ∅
6:     k_m = 0
7:     for all s ∈ S,   rhs(s) = g(s) = ∞
8:     rhs(s_start) = 0
9:     U.Insert (s_start, [h(s_start); 0])
10: end procedure
11: procedure UpdateVertex(u)
12:     if (u ≠ s_goal) then
13:         rhs(u) = min_{s'∈Succ(u)}(g(s') + c(s', u))
14:     end if
15:     if (u ∈ U) then
16:         U.Remove(u)
17:     end if
18:     if (g(u) ≠ rhs(u)) then
19:         U.Insert(u, CalculateKey(u))
20:     end if
21: end procedure
22: procedure ComputeShortestPath
23:     while (U.TopKey() < CalculateKey(s_start) OR rhs(s_start) ≠ g(s_start)) do
24:         k_old = U.TopKey()
25:         u = U.Pop()
26:         if (k_old < CalculateKey(u)) then
27:             U.Insert(u, CalculateKey(u))
28:         else if g(u) > rhs(u) then
29:             g(u) = rhs(u)
30:             for all s ∈ Pred(u) do
31:                 UpdateVertex(s)
32:             end for
33:         else
34:             g(u) = ∞
35:             for all s ∈ Pred(u) ⋃ U do
36:                 UpdateVertex(s);
37:             end for
38:         end if
39:     end while
40: end procedure
41: procedure Main
42:     s_last = s_start
```

```
43:     Initialize()
44:     ComputeShortestPath()
45:     while s_start ≠ s_goal do
46:         /* if (g(s_start) = ∞) then there is no known path */
47:         s_start = argmin_{s'∈Succ(s_start)}(c(s_start, s') + g(s'))
48:         Move to s_start
49:         Scan graph for changed edge costs
50:         if any edge costs changed then
51:             k_m = k_m + h(s_last, s_start)
52:             s_last = s_start
53:             for all directed edges (u, v) with changed edge costs do
54:                 Update the edge cost c(u, v)
55:                 UpdateVertex(u)
56:             end for
57:             ComputeShortestPath()
58:         end if
59:     end while
60: end procedure
```

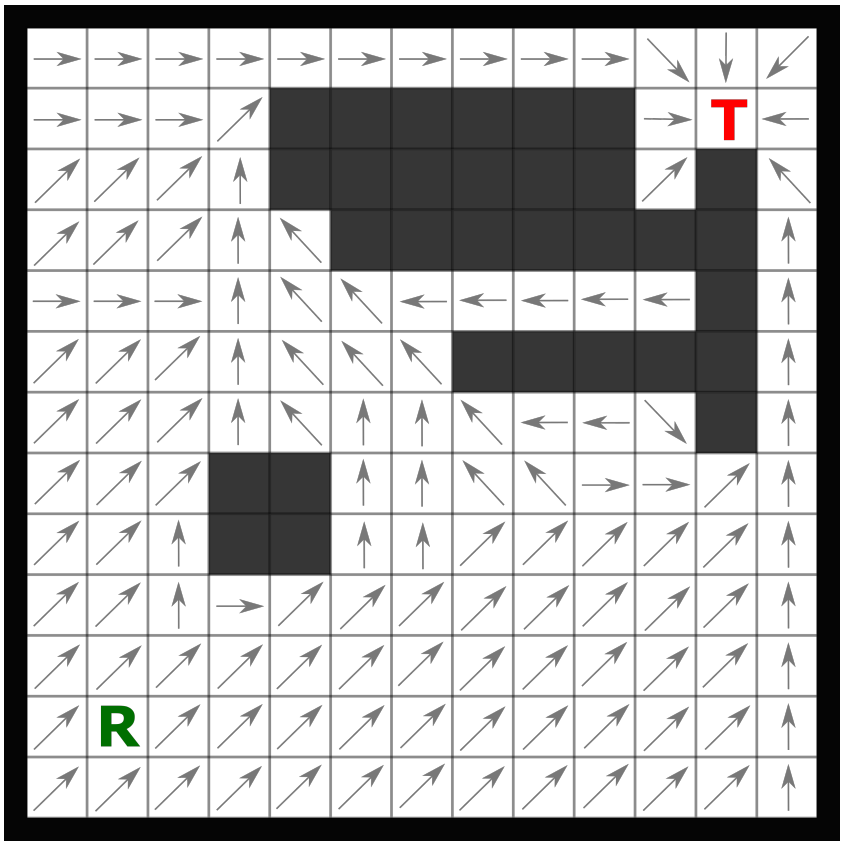


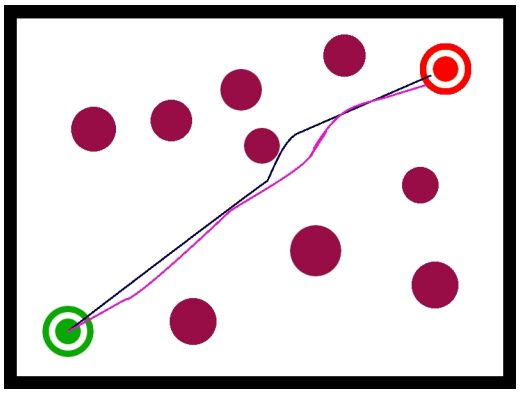
Figure 18: An updated version of Figure 17 using D* Lite. The arrows have been been updated and the blocked gate between the two obstacles has been resolved.

3.4 Experimental Results

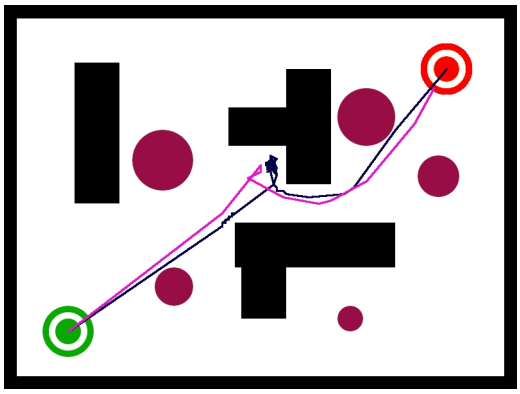
D* Lite is a graph-based path planner, so the resolution of the map is not perfect. In this section there is no optimization for either the probabilistic path planning algorithm or the D* Lite algorithm; the values of the parameters are chosen experimentally. The D* Lite algorithm was modified to include higher costs for vertices near obstacles so it could maintain some free space between the robot and the obstacles. There is no optimization involved in this set of comparisons. The number of time steps and number of collisions are recorded for both algorithms and are presented in this section.

The size of the map is 800×600 pixels. This corresponds to a map size of is 100 m $\times$ 75 m. In D* Lite the robot can see 80 pixels ahead and in the probabilistic method the robot can see 10 m ahead. The sizes of the dynamic obstacles are random and range between 56-80 pixels or 7-10 meters. The directions of their movements are random as well and they can move in any direction. The two algorithms are tested in this section on three maps as shown in Figure 19. Each algorithm was simulated three times per map because of the random sizes and movements of the obstacles, and the results are averaged. The resolution of the maps is 10 pixels. This means that the robot will jump 10 pixels per time step in both the D* Lite and probabilistic algorithms. The adaptive speed feature for the probabilistic method is disabled to provide a more even comparison.

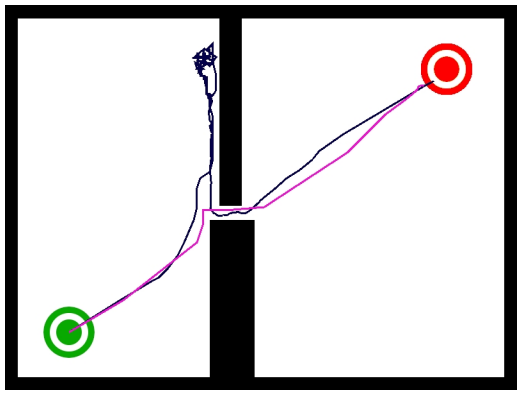
Map 1, shown in Figure 19a, contains only dynamic obstacles and no static obstacles except the surrounding walls. The blue line shows a sample path of the robot using the probabilistic approach and the red line shows a sample path using D* Lite. There are nine dynamic obstacles in the map and their direction of motion is random. We can observe that the robot paths for the two algorithms are similar. Both robots start moving toward the target point and change their paths to avoid the dynamic obstacles that appear in the visible radar radius. The speed of the dynamic obstacles



(a) Map 1



(b) Map 2



(c) Map 3

Figure 19: Maps used to compare the results for the probabilistic method and the D* Lite algorithm

are slightly higher than the robot, which makes occasional collisions inevitable.

Map 2, shown in Figure 19b, constains both static and dynamic obstacles. The robots try to avoid both the static obstacles and the dynamic obstacles. However, the robot does not which obstacles are static and which ones are dynamic. The blue line shows a sample path for the probabilistic method and the red line shows a sample path for D* Lite. D* Lite spends less time than the probabilistic method finding the path to the target.

Map 3, shown in Figure 19c, shows an environment with only static obstacles. The robot needs to find the gap in the wall and successfully navigate through it. The blue line shows a sample path for the probabilistic method and the red line shows a sample path for D* Lite.

Table IV presents the overall comparison between the two algorithms. D* Lite had better results in terms of the time required to reach the target, but the probabilistic method resulted in fewer collisions. However, we should consider that the maps were converted to a grid of vertices to make this comparison. For larger maps, or for maps with higher resolution, the probabilistic method won't have the performance penalty that D* Lite has, but D* Lite will take more time to reach the target point. D* Lite will also require more computing power as the number of nodes increases, while the probabilistic method requires computing power that is independent of the number of nodes. Overall, it appears that the probabilistic method is more suitable for real-world applications in which collisions are a primary consideration, and is a competitive path planning method for unknown environments with dynamic obstacles. In the next chapter the parameters of the robot are optimized using BBO.

	D* Lite		Probabilistic Method	
Map	Collisions	Time Step	Collisions	Time Step
Map 1	2.34	141.67	0.34	180
Map 2	1.34	171	1	211.34
Map 3	0	151	0	266.34

Table IV: Comparison between D* Lite and the probabilistic method

CHAPTER IV

BIOGEOGRAPHY-BASED OPTIMIZATION

Biography-based optimization (BBO) is an optimization algorithm based on the study of the geographical distribution of biological species. The study of biogeography dates back to the 19th century [31] when scientists began analyzing the migration behaviors of species between islands and the reasons for their extinction. BBO applies the mathematics of biogeography to engineering problems.

There are many heuristic optimization methods, such as genetic algorithms (GAs), neural networks, fuzzy logic and particle swarm optimization. Due to the advantages of BBO, which will be discussed in the following, BBO is the method chosen in this research. As a general overview, habitat suitability index (HSI) is a metric of how suitable an island is for habitation. Habitats of an island with higher HSI tend to live longer and have a higher population than habitats in islands with a lower HSI. Because of this, the population of an island with a high HSI tends to emigrate to an island with a lower HSI. Figure 20 represents a map containing different islands with population members represented by gray dots. Representatives of these members tend to move to islands with lower HSI as discussed above.

In this research, we generate a population of islands where each individual has multiple parameters that define its characteristics. To measure the "goodness" of a member of the population, a cost function is defined. This cost function evaluates a metric for each member of the population so they can be compared to each other and

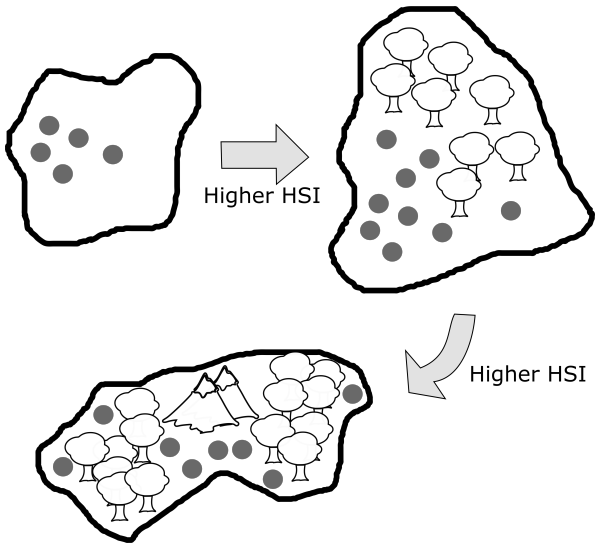


Figure 20: Biogeography migration of species to islands with lower habitat suitability index

sorted. Each iteration of the algorithm, a new population is generated via migration of parameters between members. However, a specific number of elite population members are maintained from the previous generation to prevent the loss of a good solutions to the underlying engineering problem. The number of elite members is empirical and affects how fast the algorithm converges to a population with optimized parameters.

Similar to biological species, in BBO an individual member of the population might mutate.The mutation probability is usually not high, but just as in biology a mutation might lead to an improved phenotype, in BBO mutation might lead to an improved member of the population. Mutation can also prevent the optimization process from getting stuck in a local optimum.

In our application, each member of the population includes a set of parameters to be optimized. Each parameter is initially randomly generated for each population member in the appropriate range. In a population of size n we begin BBO with n collections of randomly generated parameters. In our application, this corresponds to n robots, where each robot is simulated using its own unique set of parameters.

We define the variable T_i^g to denote the number of time steps to reach the target and C_i^g to denote the number of collisions in BBO generation g where $i \in [1, n]$ is the index of the particular member of the population. However, note that a given robot simulation will not return the same T_i^g and C_i^g values because of the randomness of the simulations; the locations and movements of the dynamic obstacles are random and vary from one simulation to the next. To quantify the average performance of the path planning algorithm for a given set of path planning parameters, we use Monte-Carlo simulations for each robot:

$$\begin{aligned} T_i^g &= \frac{T_{i1}^g + T_{i2}^g + ... + T_{iM}^g}{M} \quad i \in [1, n] \\ C_i^g &= \frac{C_{i1}^g + C_{i2}^g + ... + C_{iM}^g}{M} \quad i \in [1, n] \end{aligned} \tag{4.1}$$

where M is the number of the Monte-Carlo iterations, i indicates the index of a specific member of the BBO population, n is the population size and g is a specific BBO generation number.

The robots keep track of their values of T_i^g and C_i^g and save them in a file as soon as they reach the target point. Since we want to minimize both T_i^g and C_i^g, we consider a cost function which includes both. However, a lower number of collisions has a higher priority than the time to reach the target. Therefore, our first step is to sort the population based on T_i^g and divide it into two sections based on an experimental threshold Q. Then, all members that have $T_i^g < Q$, are sorted based on C_i^g. In this way we obtain a list of members that are ordered starting from the best (lowest) C_i^g, each of which has $T_i^g < Q$. This process is depicted in Figure 21. In the next step we choose our elite BBO members and proceed to the next generation. This continues for a desired number of generations, or until we stop achieving significant improvements. Then the algorithm reports the best member of the final population as the solution to the optimization problem.

In our simulations we set the BBO generation limit to 50 and the population

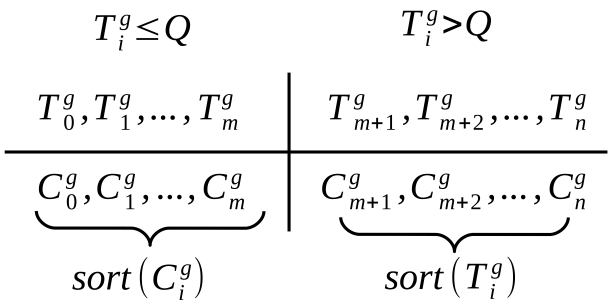


Figure 21: Depiction of how the time and collision cost functions are combined. The first m elements all have a time cost function less than a given threshold and are then sorted according to the number of collisions. The last $n-m$ elements all have a time cost function greater than the threshold and are sorted according to time.

size to 20, and each population member has 14 parameters that can be varied, as discussed in the following chapter. This setup results in a total of $50 \times 20 = 1000$ simulations for each map to generate the optimal path planning algorithm parameters. However, as discussed above, each time the map is generated, the dynamic obstacles and their direction of movement is random. So to obtain average performance of the path planning algorithm for a given set of path planning parameters, the same map is simulated for 8 Monte-Carlo iterations. Therefore, the total number of simulations is 8000 per hap. Figure 22 shows the block diagram of the BBO algorithm.

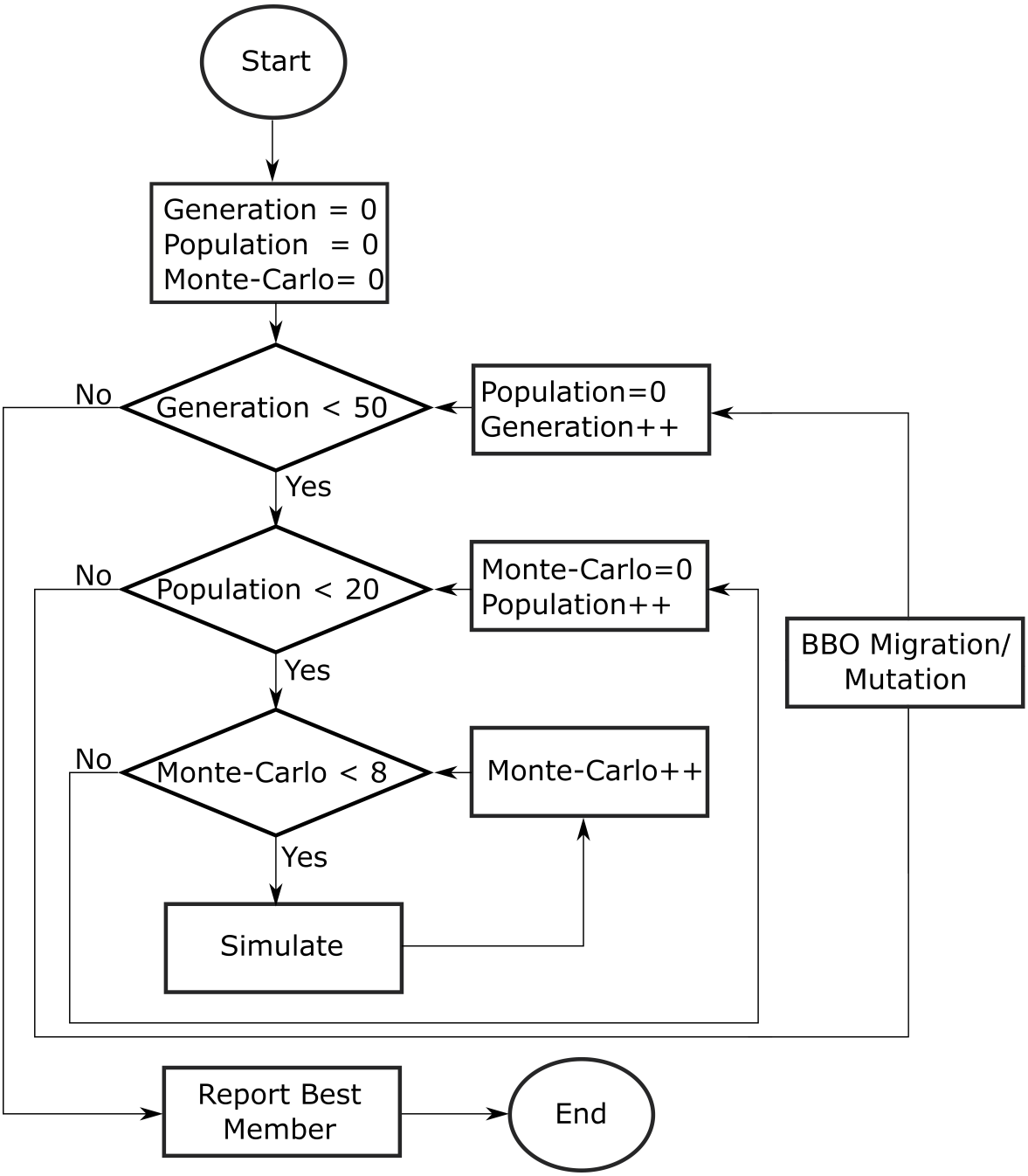


Figure 22: Block diagram of BBO algorithm

CHAPTER V

SIMULATION RESULTS

In this chapter, experimental results are presented. The simulations were performed with four maps. BBO was also implemented to optimize the probabilistic path planning algorithm parameters. In each map, the region is bounded by walls. Walls are required to prevent the robot from navigating away from the map, and also to prevent the simulation from crashing due to unknown memory access issues. The target point is depicted as a red dot and the starting point is shown as a green dot. Dynamic obstacles are blue and are defined in varying numbers and sizes for each map. The size of each map is 100×100 meters. As soon as the simulation starts, the movement directions of the dynamic obstacles are assigned randomly. In all simulations the robot starts from the lower left corner of the map and navigates to the target point, which is located at the top right corner of the map. In each section in this chapter a table shows the BBO results.

5.1 Only Dynamic Obstacles, No Bouncing after Collisions

The first set of maps contain no static obstacles other than the surrounding walls. The number of dynamic obstacles in this example is 100 and the radius of each of these obstacles varies between 1 and 4 meters. In this map, Figure 23, there is no bouncing after collisions. This means that if the robot hits an obstacle or if two obstacles hit each other, they won't bounce. Instead, based on their direction of

movement, they might stick to each other; or if an obstacles hits the robot, it might stick to the robot until another obstacle hits the first obstacle to separate it from the robot. In general, in order for the robot and obstacle to separate from each other after a collision, one of them needs to change its direction of movement by receiving an external force. Figure 24 shows the distance from the robot to the target point as a function of time for a sample simulation. Figure 25 shows the speed of the robot as a function of time for a sample simulation. The robot speed changes in order to avoid collisions, as shown in Figure 10.

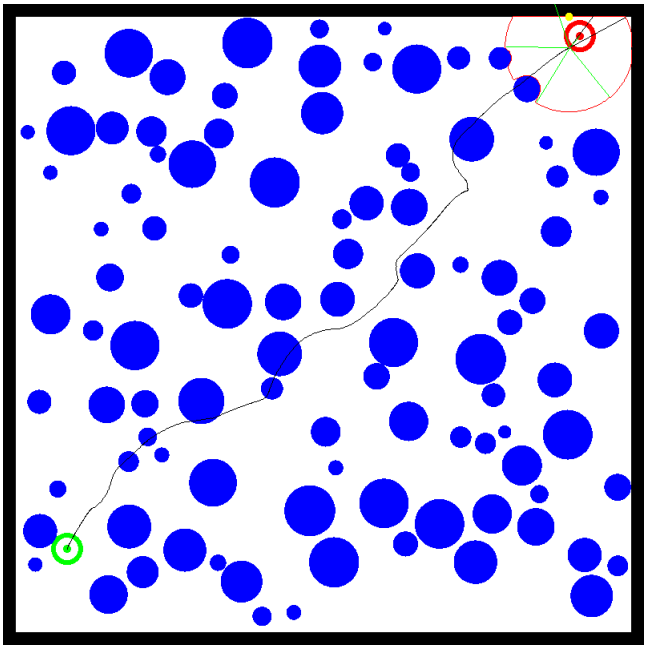


Figure 23: This map shows the robot and a sample trajectory from the starting point to the target point. The map contains only dynamic obstacles. In this map there is no bouncing after collisions.

BBO was also simulated on the sample map of Figure 23. BBO was performed for 50 generations, in each generation there were 20 members and each member was simulated with 8 Monte Carlo iterations. This resulted in a total of 8000 simulations to optimize the path planning algorithm parameters. The optimized parameters are shown in Table V.

Figures 26 and 27 show that as BBO progresses through successive generations,

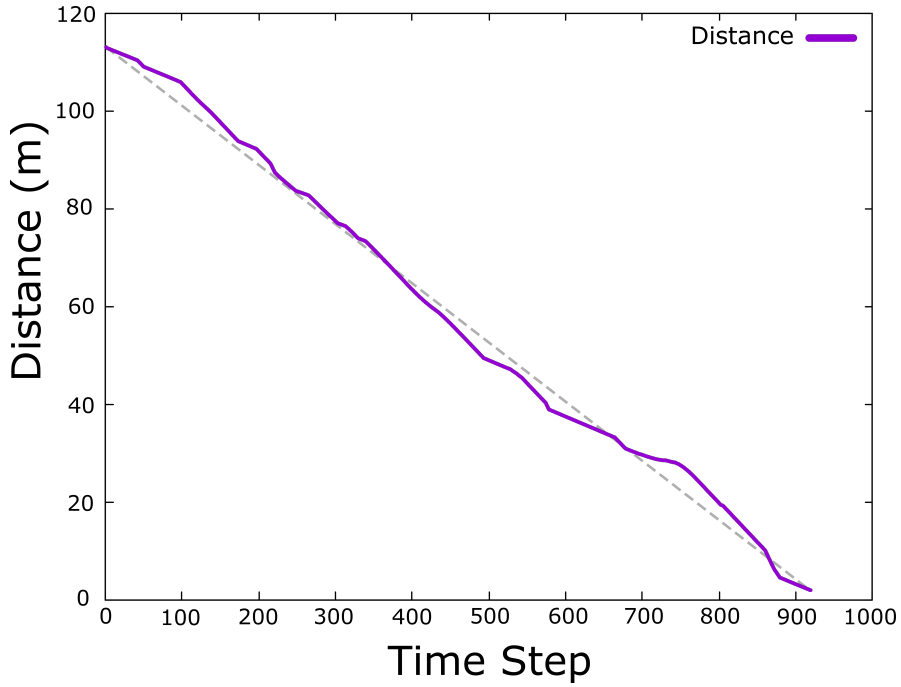


Figure 24: Distance from the robot to the target as a function of time for a sample simulation of the map of Figure 23.

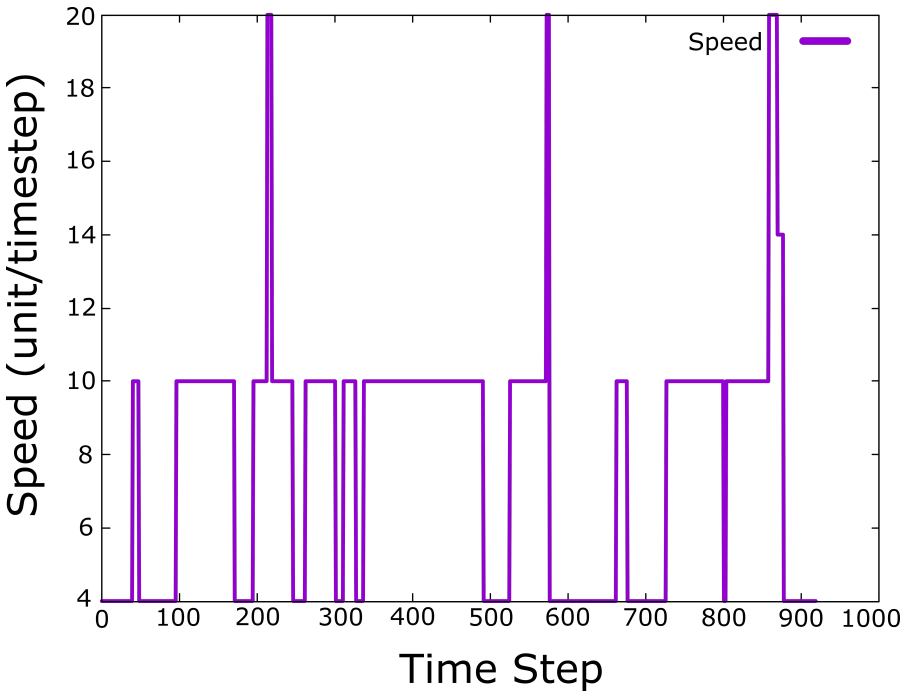


Figure 25: Robot speed as a function of time for a sample simulation of the map of Figure 23.

Variable	Range	Optimized Value	Units
Target Distribution Sigma	[30.0, 100.0]	99.16	
Memory Distribution Sigma	[70.0, 120.0]	91.88	
Very Slow Speed	[0.0, 2.0]	0.3	Units/TS
Slow Speed	[2.0, 5.0]	2.02	Units/TS
Normal Speed	[5.0, 7.0]	6.86	Units/TS
Fast Speed	[8.0, 10.0]	9.99	Units/TS
Very Fast Speed	[10.0, 20.0]	15.71	Units/TS
Corner Distance Threshold	[10.0, 50.0]	43.02	
Final Target Vector Weight	[1.0, 2.0]	1.71	
Memory Vector Weight	[0.0, 0.25]	0.24	
Front View Angle θ_2	[0.0, 60.0]	36	Degrees
Back View Angle θ_1	[θ_2, θ_2 + 60]	60	Degrees
Slow Speed Region Radius	[0, 0.99]	0.11	
Acceleration Speed Region Radius	[0, 0.99]	0.89	

Table V: Optimized parameters for the map of Figure 23. TS stands for time step.

the number of collisions and the number of time steps required to reach the target decreases. When the number of time steps reaches about 105, which is the threshold of the cost function discussed in Chapter IV, we see some unusual behavior in the number of collisions. This is because of the method of generating the cost function. As the number of time steps required to reach the target decreases to the cost function threshold, a BBO member that is greater than the threshold decreases and becomes the best member in the population, even though its collision count is greater than the previous generation's best member.

5.2 Only Dynamic Obstacles, With Bouncing

This map is the same as Figure 23; there are no static obstacles other than the surrounding walls. There are 100 dynamic obstacles and their radii vary between 1 and 4 meters. However, bouncing is enabled in the simulation. This means the obstacles will bounce away from the robot, wall, or other obstacle after they collide. A physics engine is used in the simulation to accurately calculate the reaction of the obstacles after collisions. The map is shown in 28.

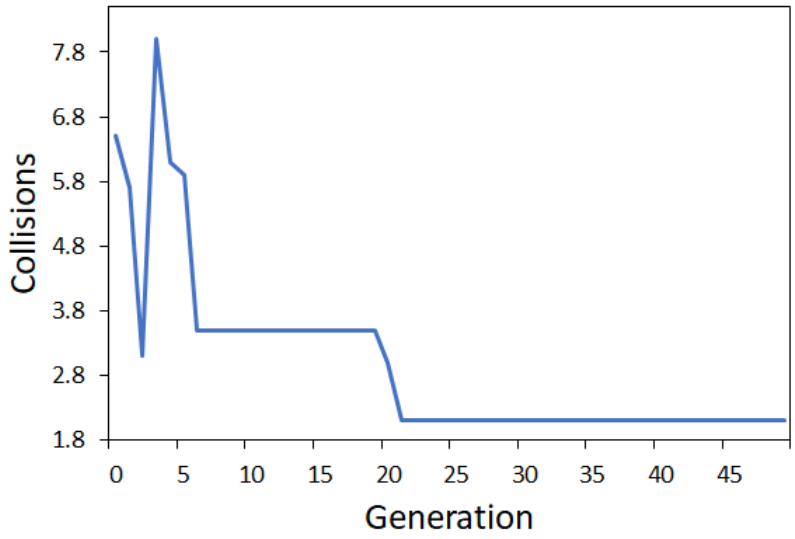


Figure 26: The number of collisions for Figure 23 improves by optimizing the path planning algorithm parameters with BBO. The number of collisions is averaged over eight Monte Carlo simulations.

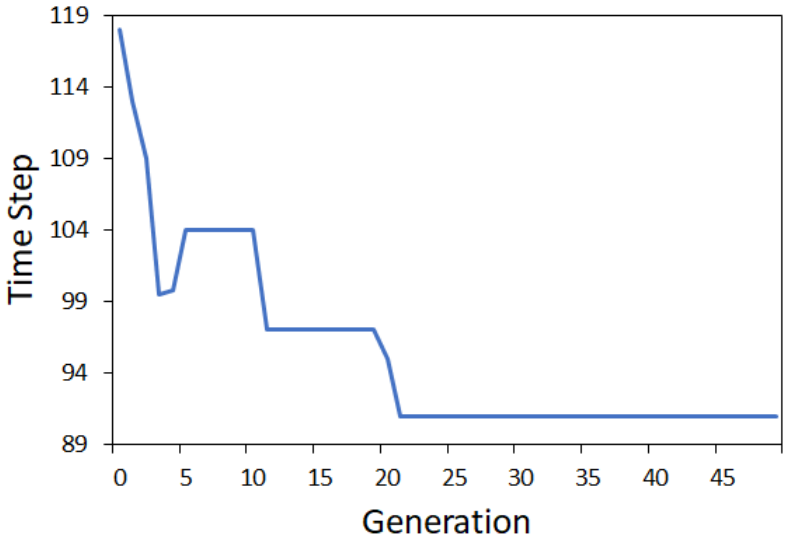


Figure 27: The number of time steps required to reach the target in Figure 23 improves by optimizing the path planning algorithm parameters with BBO. The number of collisions is averaged over eight Monte Carlo simulations.

Figure 29 shows the distance from the robot to the target point as a function of time for a sample simulation. Figure 30 shows the speed of the robot as a function of time for a sample simulation. The robot speed changes in order to avoid collisions,

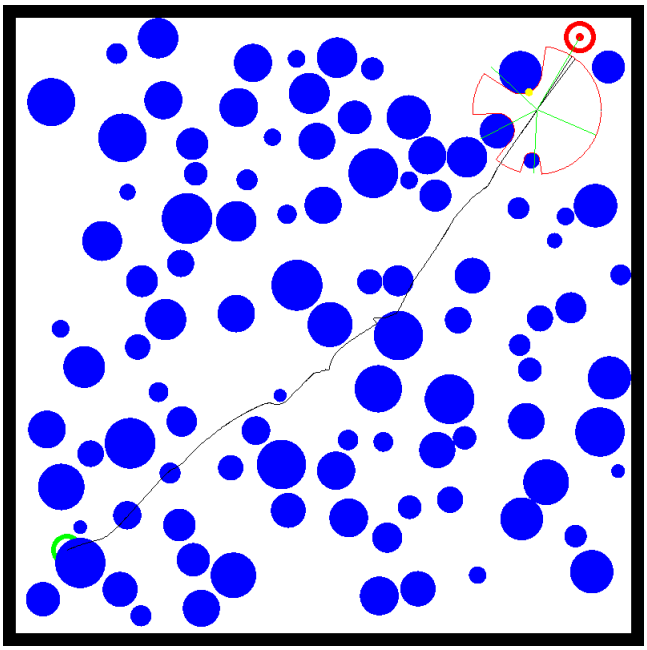


Figure 28: This map shows the robot and a sample trajectory from the starting point to the target point. The map contains only dynamic obstacles. In this map there is bouncing after collisions.

as shown in Figure 10.

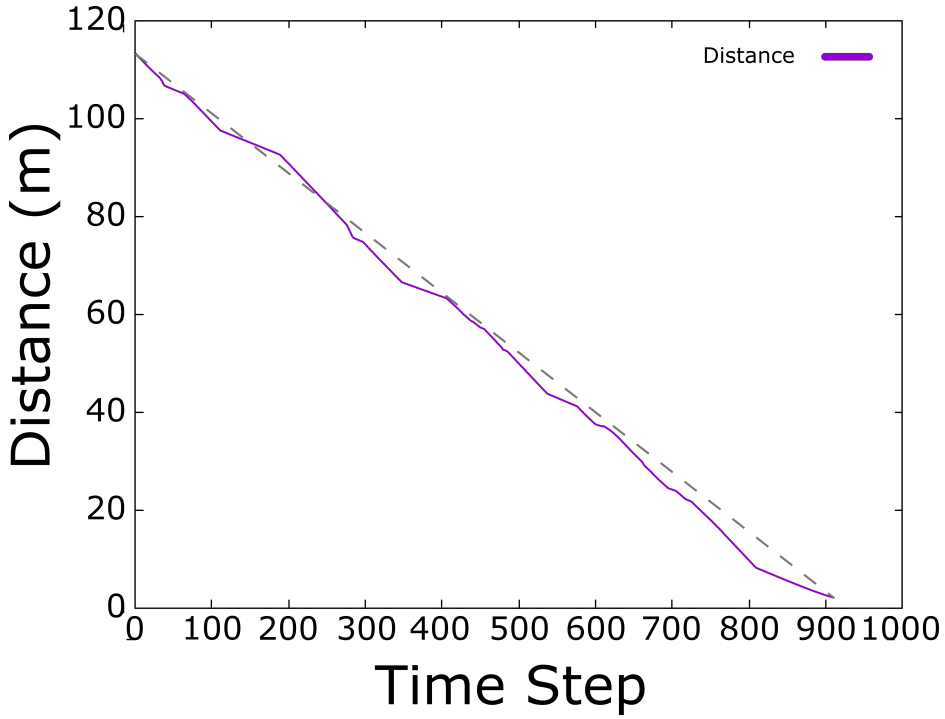


Figure 29: Distance from the robot to the target as a function of time for a sample simulation of the map of Figure 28.

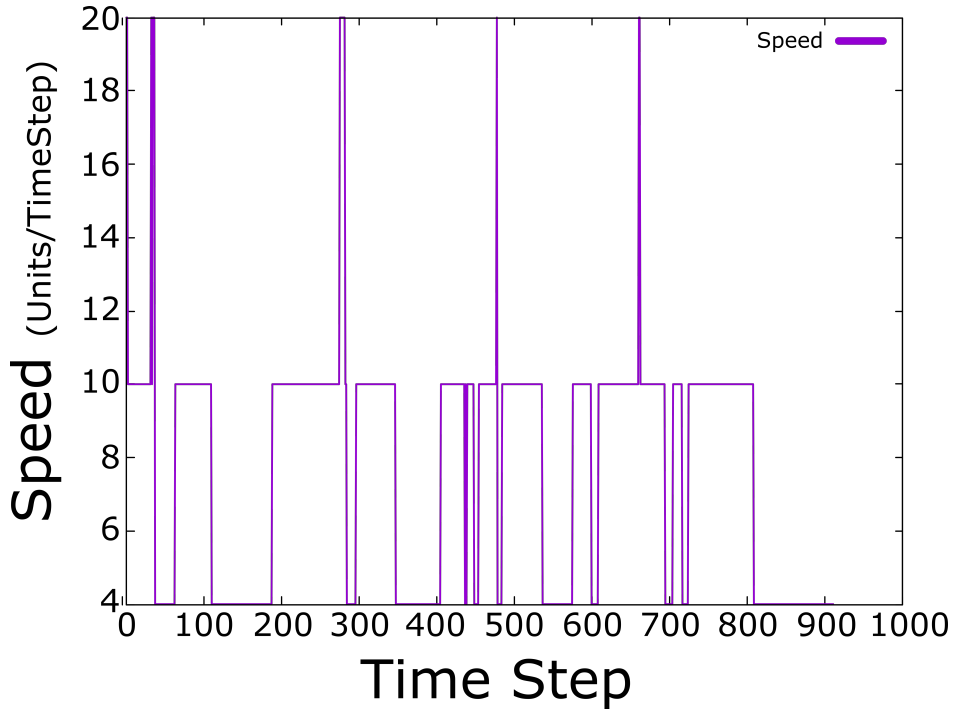


Figure 30: Robot speed as a function of time for a sample simulation of the map of Figure 28.

BBO was also simulated on the sample map of Figure 28. BBO was performed for 50 generations, in each generation there were 20 members and each member was simulated with 8 Monte Carlo iterations. This resulted in a total of 8000 simulations to optimize the path planning algorithm parameters. The optimized parameters are shown in Table VI.

As in the previous simulation, Figures 31 and 32 show that as BBO progresses through successive generations, the number of collisions and the number of time steps required to reach the target decreases. When the number of time steps reaches about 105, which is the threshold of the cost function discussed in Chapter IV, we see some unusual behavior in the number of collisions. This is because of the method of generating the cost function. As the number of time steps required to reach the target decreases to the cost function threshold, a BBO member that is greater than the threshold decreases and becomes the best member in the population, even though its collision count is greater than the previous generation's best member. The difference

Variable	Range	Optimized Value	Unit
Target Distribution Sigma	[30.0, 100.0]	98.12	
Memory Distribution Sigma	[70.0, 120.0]	80.28	
Very Slow Speed	[0.0, 2.0]	0.2	Units/TS
Slow Speed	[2.0, 5.0]	2.00	Units/TS
Normal Speed	[5.0, 7.0]	6.5	Units/TS
Fast Speed	[8.0, 10.0]	9.99	Units/TS
Very Fast Speed	[10.0, 20.0]	19.43	Units/TS
Corner Distance Threshold	[10.0, 50.0]	41.00	
Final Target Vector Weight	[1.0, 2.0]	1.91	
Memory Vector Weight	[0.0, 0.25]	0.17	
Front View Angle θ_2	[0.0, 60.0]	38	Degrees
Back View Angle θ_1	[θ_2, θ_2 + 60]	53	Degrees
Slow Speed Region Radius	[0, 0.99]	0.04	
Acceleration Speed Region Radius	[0, 0.99]	0.96	

Table VI: Optimized parameters for the map of Figure 28. TS stands for time step.

between this simulation and the previous one is that in this one, bouncing is enabled. This results in more collisions because the bouncing behavior of the obstacles can cause them to hit the robot multiple times.

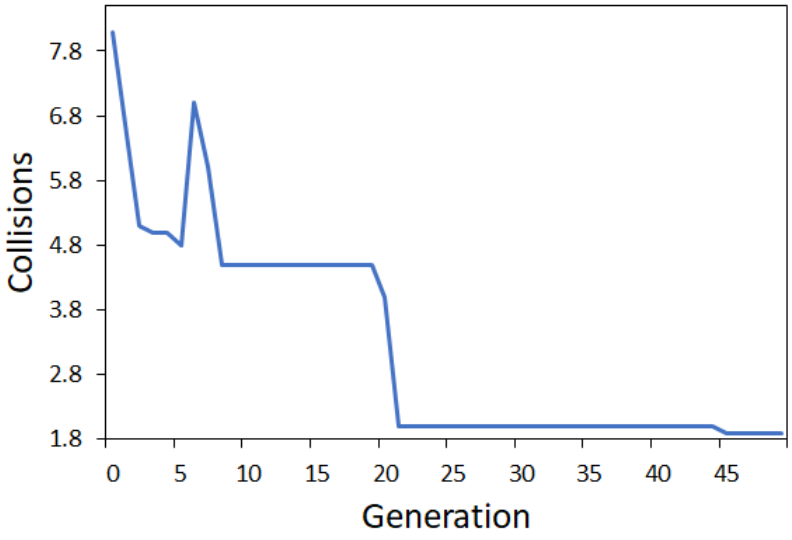


Figure 31: The number of collisions for Figure 28 improves by optimizing the path planning algorithm parameters with BBO. The number of collisions is averaged over eight Monte Carlo simulations.

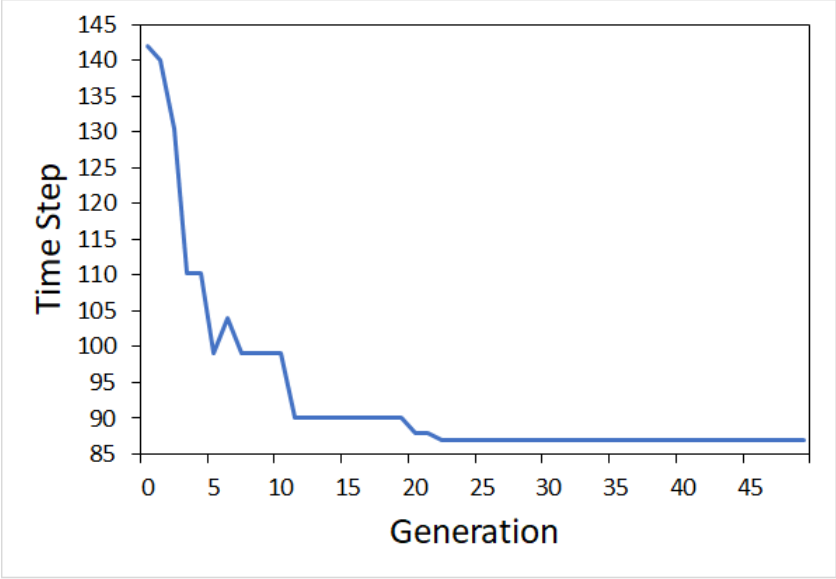


Figure 32: The number of time steps required to reach the target in Figure 28 improves by optimizing the path planning algorithm parameters with BBO. The number of collisions is averaged over eight Monte Carlo simulations.

5.3 Simple Maze with Dynamic Obstacles

In this simulation the behavior of the robot is tested in a simple maze map. The bouncing feature of the dynamic obstacles is activated. There are 50 dynamic obstacles and their sizes range between 1 and 3 meters. The map is shown in 33.

Figure 34 shows the distance from the robot to the target point as a function of time for a sample simulation. Figure 35 shows the speed of the robot as a function of time for a sample simulation. The robot speed changes in order to avoid collisions, as shown in Figure 10.

BBO was also simulated on the sample map of Figure 33. BBO was performed for 50 generations, in each generation there were 20 members and each member was simulated with 8 Monte Carlo iterations. This resulted in a total of 8000 simulations to optimize the path planning algorithm parameters. The optimized parameters are shown in Table VII.

As in the previous simulations, Figures 36 and 37 show that as BBO progresses

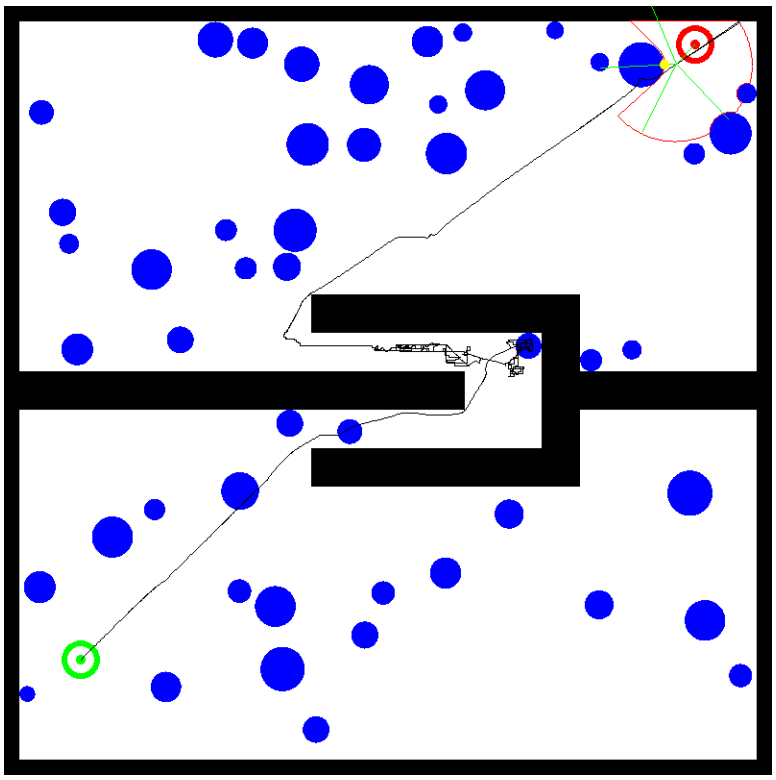


Figure 33: This map shows the robot and a sample trajectory from the starting point to the target point. The map contains only dynamic obstacles. Bouncing after collisions is enabled.

Variable	Range	Optimized Value	Unit
Target Distribution Sigma	[30.0, 100.0]	99.97	
Memory Distribution Sigma	[70.0, 120.0]	111.23	
Very Slow Speed	[0.0, 2.0]	0.01	Units/TS
Slow Speed	[2.0, 5.0]	2.00	Units/TS
Normal Speed	[5.0, 7.0]	6.88	Units/TS
Fast Speed	[8.0, 10.0]	9.67	Units/TS
Very Fast	[10.0, 20.0]	19.12	Units/TS
Corner Distance Threshold	[10.0, 50.0]	26.00	
Final Target Vector Weight	[1.0, 2.0]	1.98	
Memory Vector Weight	[0.0, 0.25]	0.23	
Front View Angle θ_2	[0.0, 60.0]	58	Degrees
Back View Angle θ_1	[θ_2, θ_2 + 60]	36	Degrees
Slow Speed Region Radius	[0, 0.99]	0.23	
Acceleration Speed Region Radius	[0, 0.99]	0.67	

Table VII: Optimized parameters for the map of Figure 33. TS stands for time step.

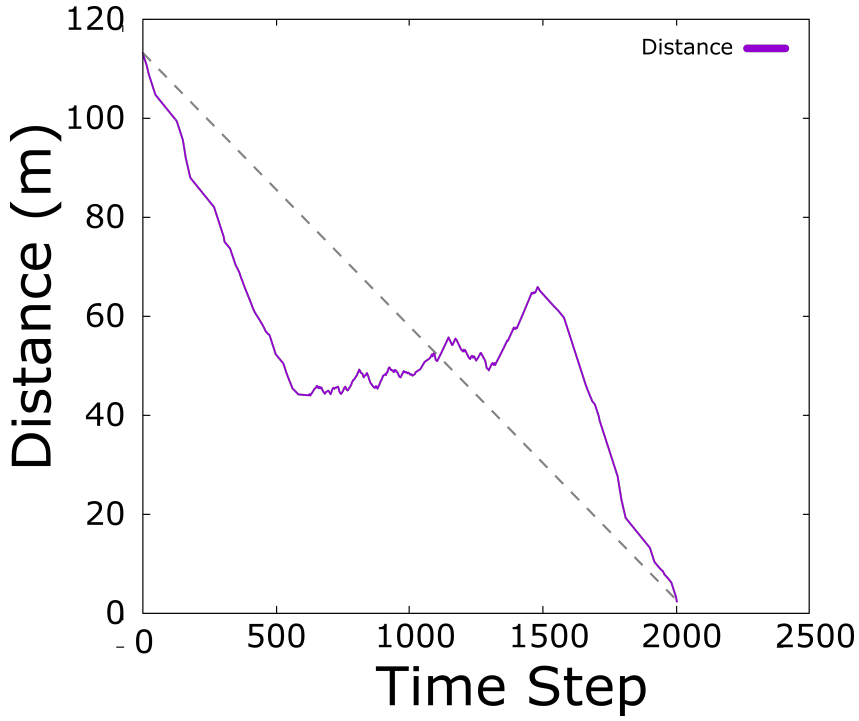


Figure 34: Distance from the robot to the target as a function of time for a sample simulation of the map of Figure 33.

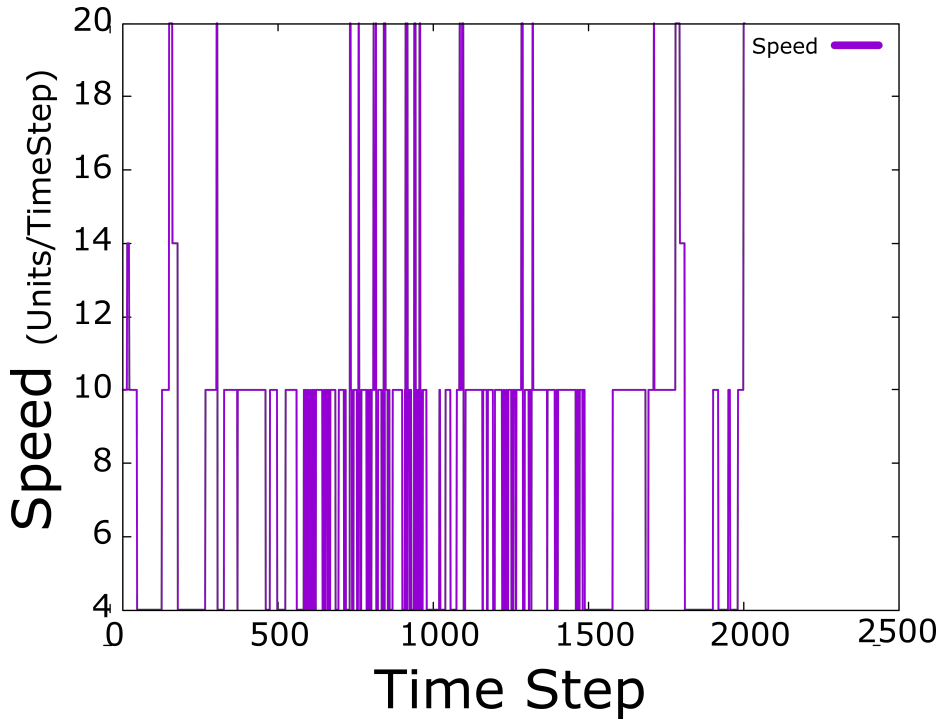


Figure 35: Robot speed as a function of time for a sample simulation of the map of Figure 33.

through successive generations, the number of collisions and the number of time steps required to reach the target decreases. The cost function threshold from Chapter IV was changed to 1500 for this map based on trial and error. Figure 37 shows that the distance to the target first decreases rapidly; however, when the robot needs to backtrack through the maze, the distance increases. When the number of time steps reaches about 1500, which is the threshold of the cost function discussed in Chapter IV, we see some unusual behavior in the number of collisions. This is similar to the behavior of the collision graphs in the previous simulations, although the threshold is larger in this simulation.

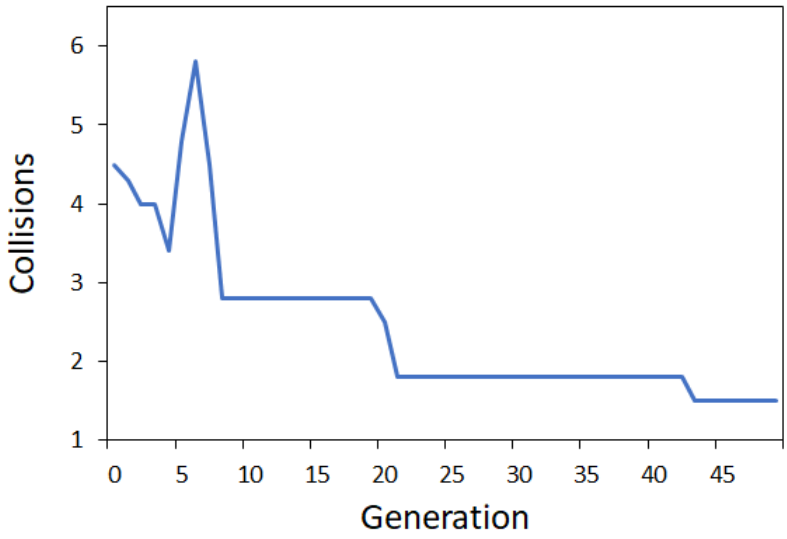


Figure 36: The number of collisions for Figure 33 improves by optimizing the path planning algorithm parameters with BBO. The number of collisions is averaged over eight Monte Carlo simulations.

5.4 Map with Rooms

In this simulation, which is the hardest map, the robot tries to reach the target point by escaping from local optima by using its memory, as discussed in Section 2.5. The memory size is 500 and bouncing between obstacles is enabled. There are 15 dynamic obstacles with radii between 4 and 5 meters. The map is shown in 38.

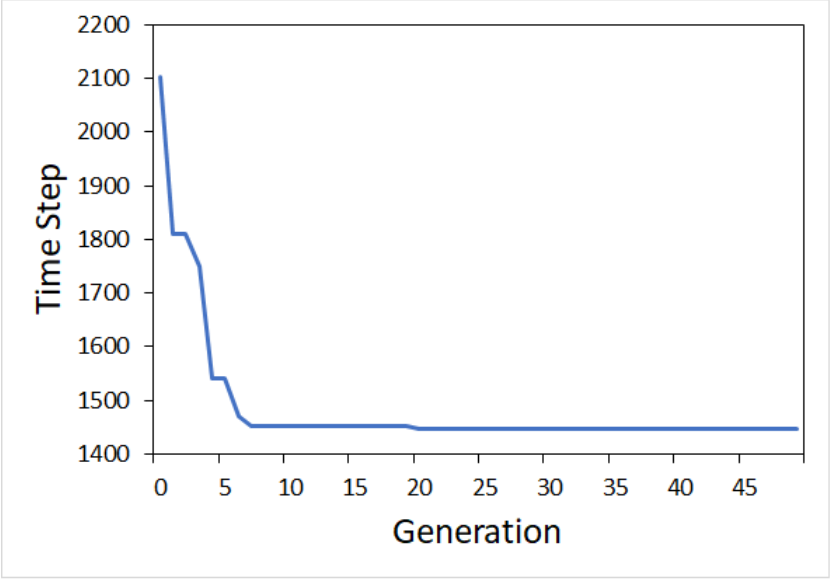


Figure 37: The number of time steps required to reach the target in Figure 33 improves by optimizing the path planning algorithm parameters with BBO. The number of collisions is averaged over eight Monte Carlo simulations.

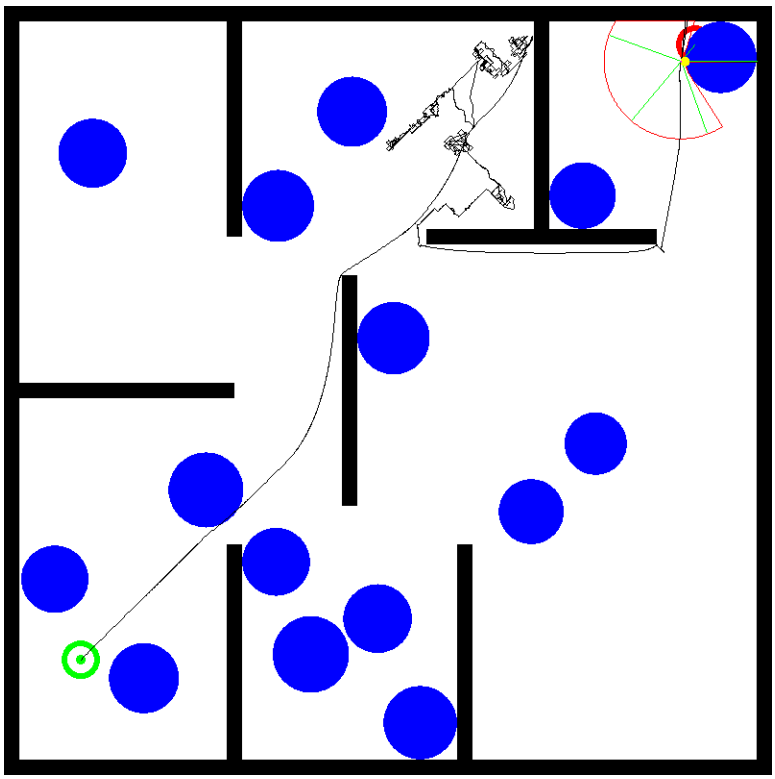


Figure 38: This map shows the robot and a sample trajectory from the starting point to the target point. The map contains several rooms in which the robot tends to get stuck. Bouncing after collisions is enabled.

Figure 39 shows the distance from the robot to the target point as a function of time for a sample simulation. Figure 40 shows the speed of the robot as a function of time for a sample simulation. The robot speed changes in order to avoid collisions, as shown in Figure 10.

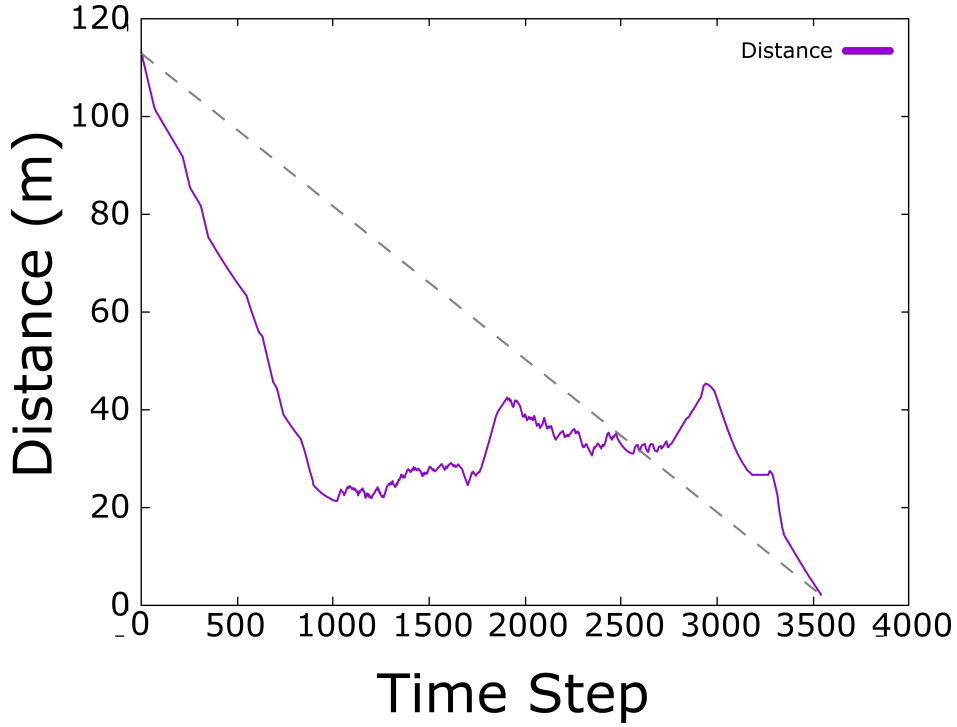


Figure 39: Distance from the robot to the target as a function of time for a sample simulation of the map of Figure 38.

BBO was also simulated on the sample map of Figure 38. BBO was performed for 50 generations, in each generation there were 20 members and each member was simulated with 8 Monte Carlo iterations. This resulted in a total of 8000 simulations to optimize the path planning algorithm parameters. The optimized parameters are shown in Table VIII.

As in the previous simulations, Figures 41 and 42 show that as BBO progresses through successive generations, the number of collisions and the number of time steps required to reach the target decreases. The cost function threshold from Chapter IV was changed to 3200 for this map based on trial and error. Figure 42 shows that

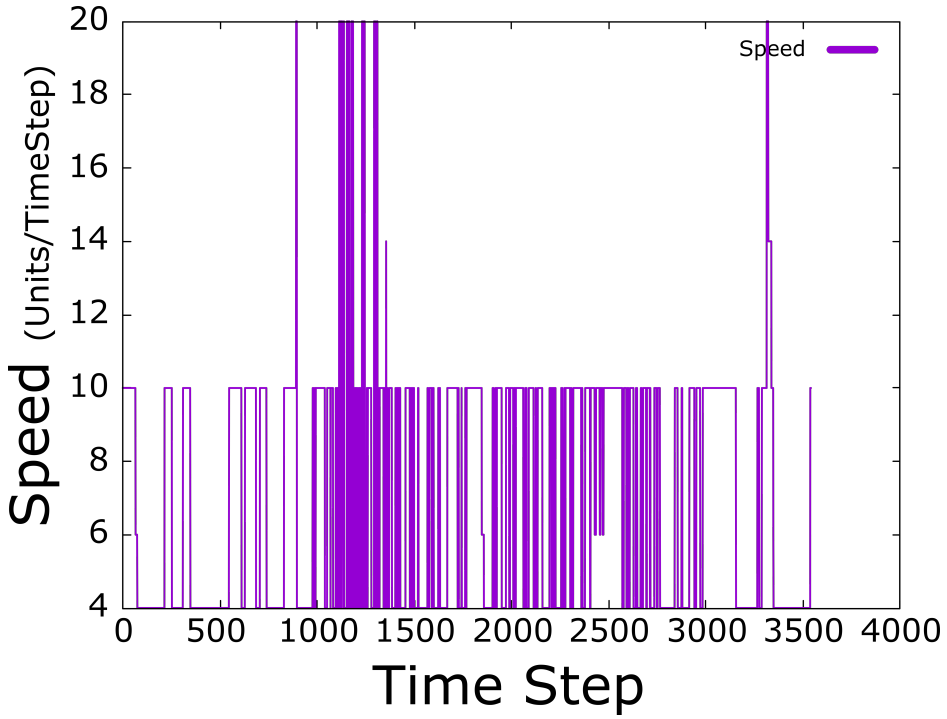


Figure 40: Robot speed as a function of time for a sample simulation of the map of Figure 38.

Variable	Range	Optimized Value	Unit
Target Distribution Sigma	[30.0, 100.0]	99.99	
Memory Distribution Sigma	[70.0, 120.0]	116.44	
Very Slow Speed	[0.0, 2.0]	0.4	Units/TS
Slow Speed	[2.0, 5.0]	3.1	Units/TS
Normal Speed	[5.0, 7.0]	6.51	Units/TS
Fast Speed	[8.0, 10.0]	9.5	Units/TS
Very Fast	[10.0, 20.0]	16.12	Units/TS
Corner Distance Threshold	[10.0, 50.0]	14.6	
Final Target Vector Weight	[1.0, 2.0]	1.6	
Memory Vector Weight	[0.0, 0.25]	0.22	
Front View Angle θ_2	[0.0, 60.0]	59	Degrees
Back View Angle θ_1	[θ_2, θ_2 + 60]	47	Degrees
Slow Speed Region Radius	[0, 0.99]	0.32	
Acceleration Speed Region Radius	[0, 0.99]	0.68	

Table VIII: Optimized parameters for the map of Figure 38. TS stands for time step.

the distance to the target first decreases rapidly; however, when the robot needs to backtrack through the maze, the distance increases. When the number of time steps reaches about 3200, which is the threshold of the cost function discussed in

Chapter IV, we see some unusual behavior in the number of collisions. This is similar to the behavior of the collision graphs in the previous simulations, although the threshold is larger in this simulation.

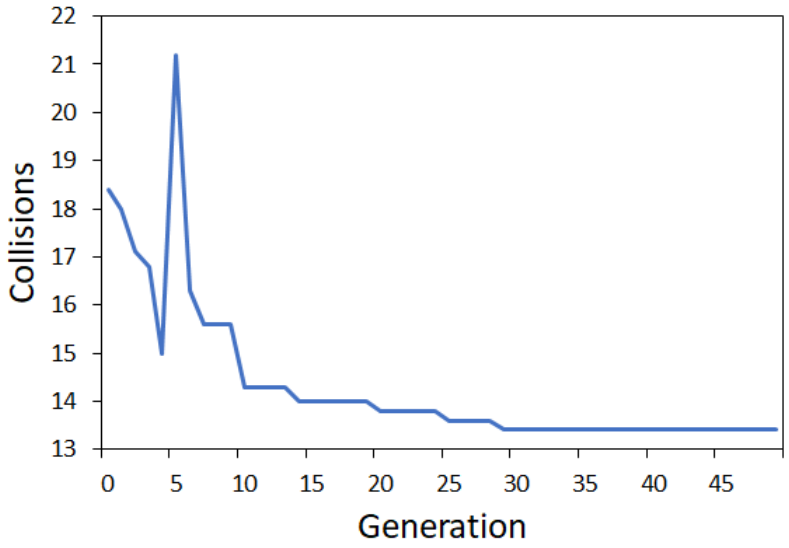


Figure 41: The number of collisions for Figure 38 improves by optimizing the path planning algorithm parameters with BBO. The number of collisions is averaged over eight Monte Carlo simulations.

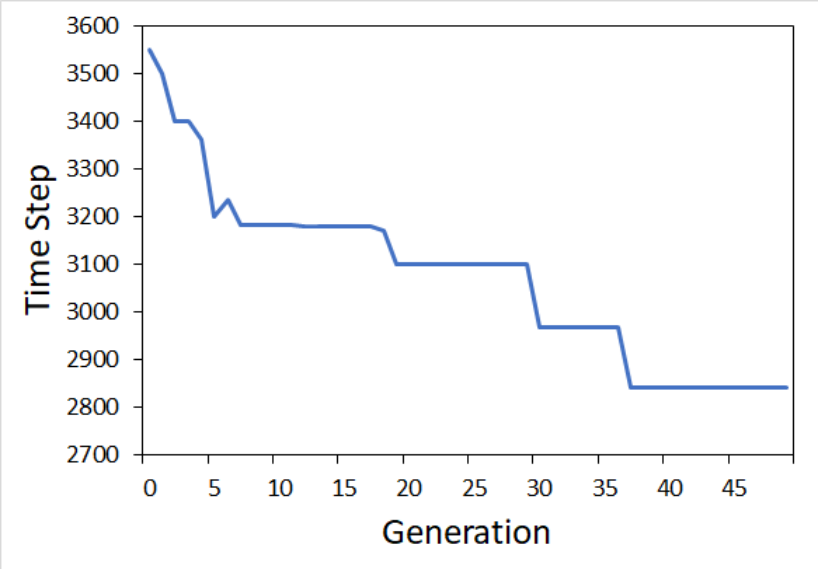


Figure 42: The number of time steps required to reach the target in Figure 38 improves by optimizing the path planning algorithm parameters with BBO. The number of collisions is averaged over eight Monte Carlo simulations.

CHAPTER VI

CONCLUSION

In this thesis, we developed a probabilistic path planning algorithm for robot navigation, simulated the probabilistic method on four different types of maps, and used BBO to optimize the parameters of the algorithm. The BBO cost function was defined so that the robot would reach its target in an acceptable period of time while minimizing the number of collisions. A memory algorithm was used to force the robot to backtrack in case it got stuck in a corner or in a room. The algorithm also includes a speed regulator so that the robot can adjust its speed to avoid collisions with dynamic obstacles while still reaching the target as quickly as possible.

Simulation results show that BBO's adjustment of the path planning algorithm parameters decreases the number of collisions and time steps by about 25%. Furthermore, the path planning algorithm is fast enough for real-time implementation and could solve all the maps in this thesis. However, path planning performance could possibly be improved further by continually running the optimization algorithm, a research direction which has not been pursued in this thesis.

Due to the randomness of the environment, the robot can react suddenly to changes in the environment and vary its diretion of movement abruptly. This could be undesirable for real robots that may be sensitive to sudden movements, or to humans in the environment that may likewise be sensitive to sudden movements by robots in their environment. For future work, reducing the abruptness of the robot's movement

fluctuations could be considered. Typical robots usually have a considerable amount of memory, but one of the goals of this thesis was to use as little robot memory as possible. To improve the performance of the robot, more memory could be used in future versions of this algorithm, especially to help the robot escape rooms and dead ends. Implementing this method on real robot hardware could be another future direction for this research.

BIBLIOGRAPHY

[1] Piotr Bigaj and Jakub Bartoszek. Low Time Complexity Collision Avoidance Method for Autonomous Mobile Robots. In D. Filev et al., editor, Intelligent Systems' 2014, pages 141–152. Springer, 2015.

[2] Terrance E Boult. Updating Distance Maps when Objects Move. In Mobile Robots II, volume 852, pages 232–240. International Society for Optics and Photonics, 1987.

[3] John Canny. The Complexity of Robot Motion Planning. MIT press, 1988.

[4] Erin Catto. Box2d: A 2d Physics Engine for Games. http://box2d.org/, 2011.

[5] Bruce A Conway. A Survey of Methods Available for the Numerical Optimization of Continuous Dynamic Systems. Journal of Optimization Theory and Applications, 152(2):271–306, 2012.

[6] Juan Cortés, Thierry Siméon, V Ruiz de Angulo, David Guieysse, Magali Remaud-Siméon, and Vinh Tran. A Path Planning Approach for Computing Large-amplitude Motions of Flexible Molecules. Bioinformatics, 21(suppl_1):i116–i125, 2005.

[7] Shi-Gang Cui, Hui Wang, and Li Yang. A Simulation Study of A-star Algorithm for Robot Path Planning. In 16th International Conference on Mechatronics Technology, pages 506–510, 2012.

[8] Hongkai Dai, Andrés Valenzuela, and Russ Tedrake. Whole-body Motion Planning with Centroidal Dynamics and Full Kinematics. In IEEE-RAS International Conference on Humanoid Robots, pages 295–302. IEEE, 2014.

[9] R Deepu, B Honnaraju, and S Murali. Path Generation for Robot Navigation Using a Single Camera. Procedia Computer Science, 46:1425–1432, 2015.

[10] Frantiek Ducho, Dominik Huady, Martin Dekan, and Andrej Babinec. Optimal Navigation for Mobile Robot in Known Environment. In Applied Mechanics and Materials, volume 282, pages 33–38, 01 2013.

[11] Laurent Gomila. Simple and Fast Multimedia Library. https://www.sfml-dev.org/, 2010.

[12] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics, 4(2):100–107, 1968.

[13] Neil Eugene Hodge, Linda Zhixia Shi, and Mohamed B Trabia. A Distributed Fuzzy Logic Controller for an Autonomous Vehicle. Journal of Field Robotics, 21(10):499–516, 2004.

[14] Yaochu Jin and Jürgen Branke. Evolutionary Optimization in Uncertain Environments-a Survey. IEEE Transactions on Evolutionary Computation, 9(3):303–317, 2005.

[15] Tae-Koo Kang, Huazhen Zhang, Gwi-Tae Park, and Dong W Kim. Ego-motion-compensated Object Recognition Using Type-2 Fuzzy Set for a Moving Robot. Neurocomputing, 120:130–140, 2013.

[16] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic Roadmaps for Path Planning in High-dimensional Configuration Spaces. IEEE Transactions on Robotics and Automation, 12(4):566–580, 1996.

[17] Do-Hyeon Kim, Kwang-Baek Kim, and Eui-Young Cha. Fuzzy Truck Control Scheme for Obstacle Avoidance. Neural Computing and Applications, 18(7):801–811, 2009.

[18] Jonas Koenemann, Andrea Del Prete, Yuval Tassa, Emanuel Todorov, Olivier Stasse, Maren Bennewitz, and Nicolas Mansard. Whole-body Model-predictive Control Applied to the Hrp-2 Humanoid. In IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 3346–3351, 2015.

[19] Sven Koenig and Maxim Likhachev. Dˆ* Lite. AAAI Conference on Artificial Intelligence, 476-483, 2002.

[20] Sven Koenig and Maxim Likhachev. Improved Fast Replanning for Robot Navigation in Unknown Terrain. In Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on, volume 1, pages 968–975. IEEE, 2002.

[21] Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong Planning Aˆ*. Artificial Intelligence, 155(1-2):93–146, 2004.

[22] James J Kuffner and Steven M LaValle. Rrt-connect: An Efficient Approach to Single-query Path Planning. In IEEE International Conference on Robotics and Automation, volume 2, pages 995–1001. IEEE, 2000.

[23] Itay Lotan, Fabian Schwarzer, Dan Halperin, and Jean-Claude Latombe. Efficient Maintenance and Self-collision Testing for Kinematic Chains. In Proceedings of the Eighteenth Annual Symposium on Computational Geometry, pages 43–52. ACM, 2002.

[24] Tomás Lozano-Pérez and Michael A Wesley. An Algorithm for Planning Collision-free Paths Among Polyhedral Obstacles. Communications of the ACM, 22(10):560–570, 1979.

[25] K. Manousakis, T. McAuley, R. Morera, and J. Baras. Using Multi-objective Domain Optimization for Routing in Hierarchical Networks. In International Conference on Wireless Networks, Communications and Mobile Computing, volume 2, pages 1460–1465 vol.2, June 2005.

[26] Matthew Moore and Jane Wilhelms. Collision Detection and Response for Computer Animation. In ACM Siggraph Computer Graphics, volume 22, pages 289–298. ACM, 1988.

[27] Nils J Nilsson. Principles of Artificial Intelligence. Morgan Kaufmann, 2014.

[28] Steven Ratering and Maria Gini. Robot Navigation in a Known Environment with Unknown Moving Obstacles. Autonomous Robots, 1(2):149–165, 1995.

[29] Andrey V Savkin and Chao Wang. Seeking a Path Through the Crowd: Robot Navigation in Unknown Dynamic Environments with Moving Obstacles Based on an Integrated Environment Representation. Robotics and Autonomous Systems, 62(10):1568–1580, 2014.

[30] Jacob T Schwartz and Micha Sharir. On the Piano Movers' Problem: Iii. Coordinating the Motion of Several Independent Bodies: The Special Case of Circular Bodies Moving Amidst Polygonal Barriers. International Journal of Robotics Research, 2(3):46–75, 1983.

[31] Dan Simon. Biogeography-based Optimization. IEEE Transactions on Evolutionary Computation, 12(6):702–713, 2008.

[32] Anthony Stentz. Optimal and Efficient Path Planning for Partially-known Environments. In IEEE International Conference on Robotics and Automation, pages 3310–3317. IEEE, 1994.

[33] Anthony Stentz et al. The Focussed dˆ* Algorithm for Real-time Replanning. In International Joint Conference on Artificial Intelligence, volume 95, pages 1652–1659, 1995.

[34] Anthony Stentz and Martial Hebert. A Complete Navigation System for Goal Acquisition in Unknown Environments. Autonomous Robots, 2(2):127–145, 1995.

[35] Bing Sun, Daqi Zhu, Lisha Jiang, and Simon X Yang. A Novel Fuzzy Control Algorithm for Three-dimensional AUV Path Planning Based on Sonar Model. Journal of Intelligent and Fuzzy Systems, 26(6):2913–2926, 2014.

[36] Meng Wang and James NK Liu. Fuzzy Logic-based Real-time Robot Navigation in Unknown Environment with Dead Ends. Robotics and Autonomous Systems, 56(7):625–643, 2008.

[37] Panagiotis G Zavlangas and Spyros G Tzafestas. Industrial Robot Navigation and Obstacle Avoidance Employing Fuzzy Logic. Journal of Intelligent and Robotic Systems, 27(1-2):85–97, 2000.

[38] Alexander Zelinsky. A Mobile Robot Exploration Algorithm. IEEE Transactions on Robotics and Automation, 8(6):707–717, 1992.

APPENDIX A

Box2D Library

Box2D is a ubiquitous and free cross-platform two-dimensional physics engine designed for the C++ programming language. It was first introduced in 2006 at GDC 2006 by Erin Catto [4]. A physics engine simulates the behavior of objects to make them behave in a real-life way. Box2D has been used in many games designed for different platforms like Android, PC, iOS and Flash. Furthermore, Box2D can be used in wide range of simulation applications to simulate the physics of particles and to visualize the dynamics of the systems. As another feature, this library allows developers to build a collision detection system. In this research, Box2D was used to simulate the detection of collisions and also to simulatethe real behavior of the obstacles in the environment.

Box2D uses metric units and can handle large distance values. In our experiments it produced results for a large number of obstacles and large rooms in a considerably short time. The other feature of this library is the high accuracy of the simulation. This means that when the robot moves between two locations, we can be sure that it won't run over the top of any obstacles.

Simulating the ray-casting of a laser beam can require very expensive computational effort and a lot of CPU resources. In our experiments, we had to do 360 ray-castings at each time step. Box2D has a highly optimized algorithm for ray-casting. So using Box2D enabled us to shorten our simulation time significantly.

Box2D objects have two important features: the first is the body of the object. The body does not contain any information about the shape and appearance of the object and contains more general properties like mass, velocity and location. Of

course, knowing these properties does not give us any information about the shape of the object. The next feature of the Box2D object is called fixture. Fixture contains information about the shape of the object in more detail. Figure 43 shows the block diagram of the features of a Box2D object. Therefore, in order to create an object the Body should first be created and then the fixture of the body should be attached to it.

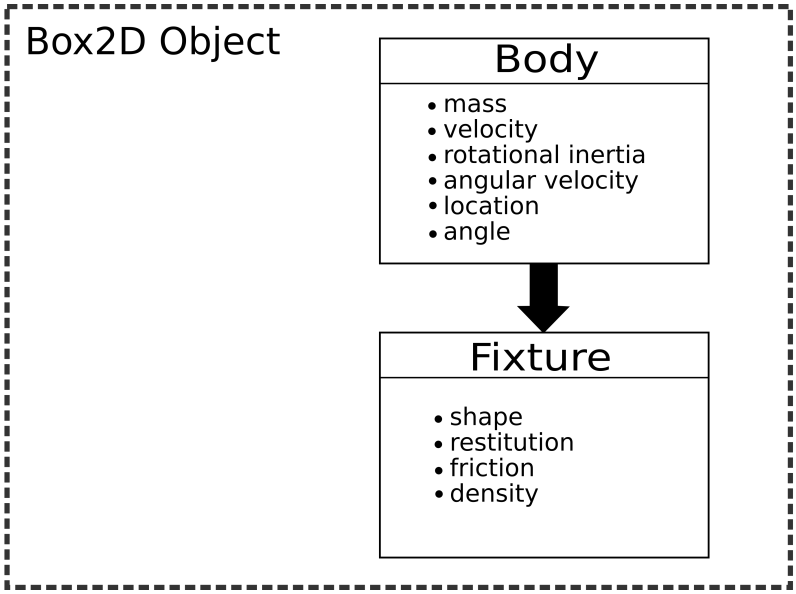


Figure 43: Overview of an object decleration in Box2D

APPENDIX B

Simple and Fast Multimedia Library

The Simple and Fast Multimedia Library (SFML) [11] is a free cross-platform library to provide a simple interface for the C++ programming language to communicate between different parts of a PC. It provides a convenient API to program applications that need to have access to network, hardware acceleration in 2D computer graphics, sound system, and the creation and input to windows with OpenGL contexts. In this research, SFML was used to create and handle OpenGL graphics windows to visualize the movements of the robot and also to debug the simulation process.

SFML is an event-based library and responds to inputs through events. Therefore, the events need to be checked in run-time so that we can take appropriate action based on the corresponding event. SFML is a multi-platform library and can be compiled for different operating systems such as Windows, Linux and Mac OS. Moreover, different bindings for SFML have been developed so that it can be used in different programming languages such as Python, Ruby, Java and .Net languages. Figure 44 shows the block diagram of the execution of the SFML in our experiments. At the first step, the window is created. Then after each time step the program checks to see if there are any events waiting to be processed in the event stack of the SFML library. If there are any events stored, the developer can react accordingly. In our experiments, the user could only interact with SFML to close the simulation through events.

Showing graphics while the simulation is being executed slows down the simulation so that the graphics become a computational bottleneck in the simulation.

To solve this issue, a graphic enable flag variable was defined so that when the optimization process is executing, graphics are not shown to the user. This method significantly increases the simulation speed.

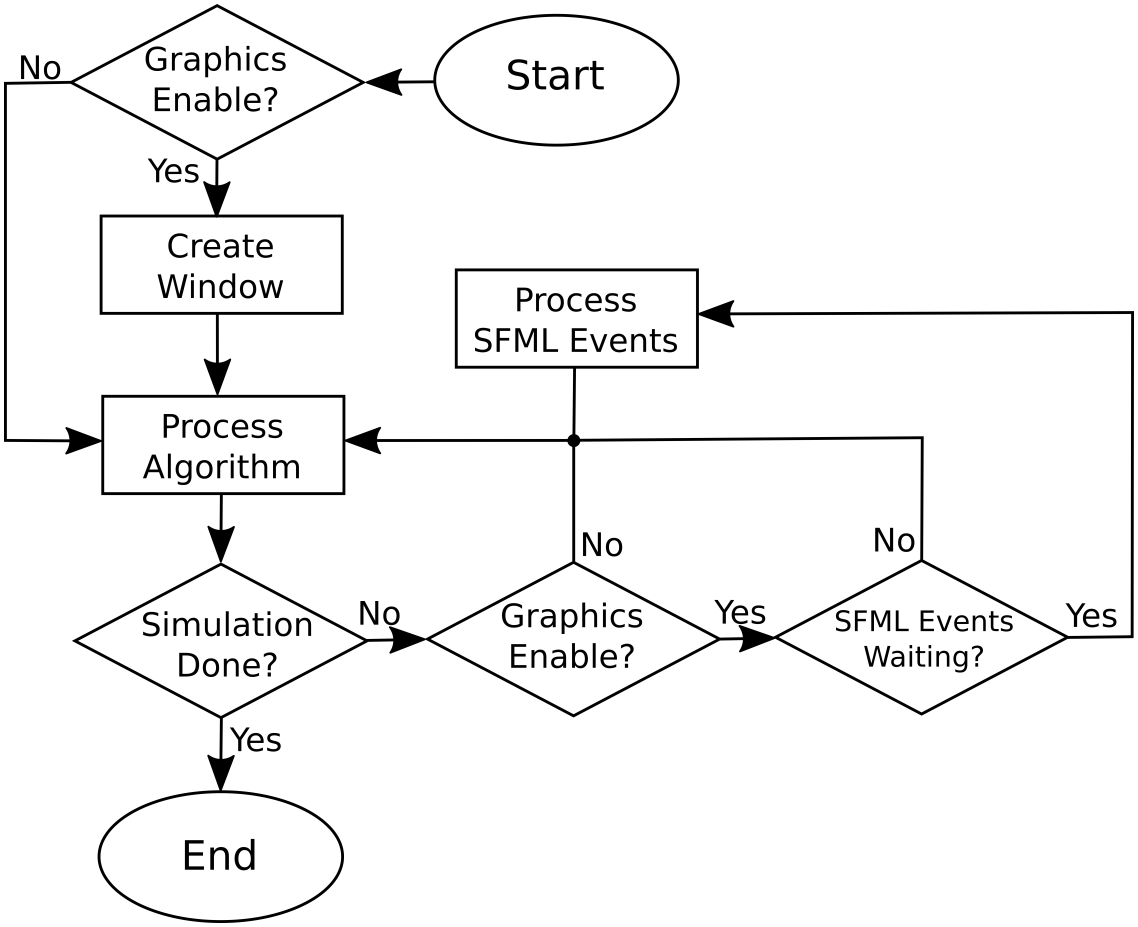


Figure 44: Block diagram of the operation of the SFML