Robot Planning - Project Report

Robot Evacuation with Adaptive Offsetting and Circle Interpolation

Authors: *Alami Ashkan mat.225087*
Barba Edoardo mat.231592
Brugnera Matteo mat.232670

Master: Artificial Intelligence Systems
Date: February 22, 2023

1 INTRODUCTION

In this project we focus on the evacuation of multiple robots from the same room, with particular emphasis on the definition of the control and coordination strategies that will be applied to perform this task in the fastest way possible.
This project discusses a class of problems related to the assembly of geometric shapes through the creation of a road-map that seeks a solution which "grazes" the obstacles. Thanks to this aggressive strategy, we are able to obtain the shortest path for each robot but we also assure the safeness of the trajectory thanks to the **adaptive offsetting** and **circle interpolation** algorithms.
As we will see in the next sections, through the combination of these two self-designed algorithms, we are able to create and interpolate a path in a efficient way.

Since the environment is *static* and *closed*, we resort to the *deliberative paradigm*. More specifically, we use a top/down philosophy where we:

1. obtain all the necessary information of the environment;
2. update the model of the environment;
3. plan a course of actions;
4. implement the plan.

In this paradigm, all the decisions are taken after a deliberation procedure and no corrections are made after this step. Even though this way of proceeding is quite slow if compared to the reactive paradigm, it gives us the chance to build a detailed model of the environment that can be then exploited to execute the coordination task smoothly.
In order to analyze all the procedures, we will take into consideration the **mission planning** and **motion planning** sections separately.

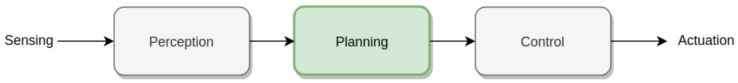


Figure 1: schema representing the deliberative paradigm.

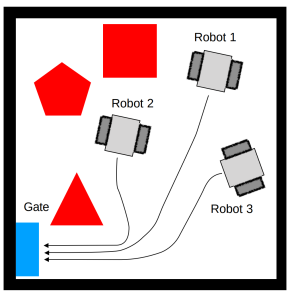


Figure 2: image representing the task.

2 MISSION PLANNING

Mission planning is the highest level task in motion planning. It is responsible for constructing the global level planning for reaching the destination.
The main objective of mission planning is *generating a sequence of actions* that we want to execute in order to achieve a certain result, starting from the specification of a *mission*. In this section we will disassemble the main problem into multiple sub-missions and we will see how we deal with each one of them (except for the creation of the trajectory which is tackled in the motion planning section).

The *main mission* is to find a route that connects a robot from its current position to an escape gate in minimum time. This has to be done without touching the walls that define the map contour and without colliding with obstacles and other robots along the way.

2.1 Obstacles detection and creation

In this scenario we have to deal with the presence of multiple obstacles which makes the execution of this task more difficult. First of all we have to detect them through some sensing and detection system.
Since we implemented this project in a *simulated environment*, we can create them directly by hard-coding the positions of the obstacles.
The system is able to retrieve all the coordinates of the obstacles thanks to the creation of an ad-hoc *topic* called *obstacles* which communicates all the coordinates of all the obstacles.

Once we have this information, we need to deal with the *collision detection issue*. It's a fundamental task for many motion planning algorithms and tools because it allows us to check if our robot is colliding with the obstacles in the map.
An effective approach to perform collision detection in this scenario is based on the application of an *offset* to the polygonal obstacles. Thanks to this, we can consider the robot as a point because the information on the geometry of the robot is already encoded by the offsetted polygons.
Instead of applying a fixed value as offset, we adapt it depending on the type of obstacles we are facing. We call this procedure **Adaptive Offsetting** and we will explain it later, in Section 3.5.

2.2 Obstacles manipulation

Once we know the *optimal offset value* for a given obstacle, we can perform its manipulation. Depending on the type of obstacle that we are dealing with, we can have different approaches. More specifically:

- if the obstacle represents the *contour of the map*, we apply an internal offsetting equal to half the robot width (to make sure that the robot doesn't collide with the borders);
- if the obstacle is a *simple polygon* laying inside the map, we perform the offsetting as explained in Section 3.5.

It can be the case that two or more obstacles *intersect* with each other after applying the offset. If this is the case we simply *merge* the polygons together. In this way we obtain a new polygon

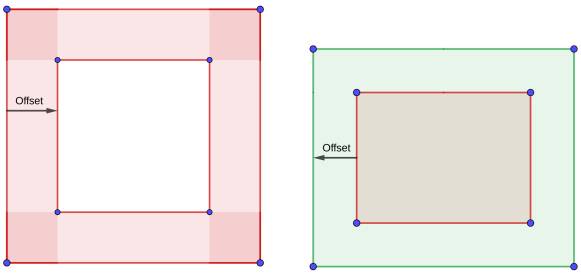


Figure 3: image representing the offset on the map borders and on a obstacle.

that substitutes the previous ones (Figure 4).
The same thing happens if an obstacle intersects with the map-contour. In this particular case, the border of the map is simply extended with the perimeter of the obstacle (Figure 5).

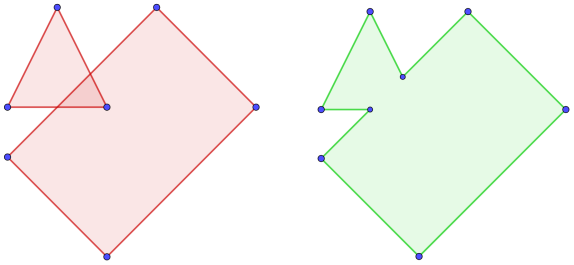


Figure 4: image representing the intersection of two obstacles and the merged resulting polygon.

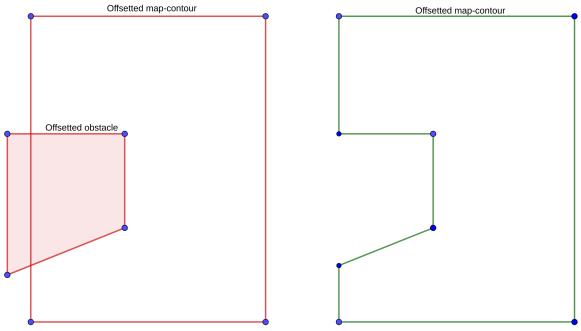


Figure 5: image representing the intersection of a polygon with the map-contour and the merged resulting border.

This way of finding out the collisions between two polygons is way more efficient than using the *two phase detection* approach or other *hierarchical methods*.
In the first case, we avoid performing the *broad phase* and we execute the *narrow phase* in a more accurate way.
In the latter case, we avoid checking in a recursive way all the different part of the collision and we also avoid building a simplification of the obstacles by defining, for instance, its *convex hull* (we will see why it's not necessary doing such in the next subsection).

Thanks to all of this (as we will see in the motion planning Section 3) we are able to obtain a *collision-free path* for every single robot even without executing other checking procedures.

2.3 Road-map creation

Now that we have all the offsetted-obstacles, we want to find an admissible collision-free path that moves the robot from its starting point to the final destination.
In order to achieve this result, we decided to resort to the combinatorial planning which allows us to build a road-map used to solve the motion planning queries. Thanks to it, we have a concise representation of the environment in form of a topological graph that can be exploited by the planner to find a path.

Among the many possible techniques that can be used to create a road-map, we opted for the **shortest-path design** also known as **Visibility Graph method**. This was done for a multitude of reason:

- *accessibility*. Once the map has been created, we can reach a point in the map from any outside point;
- *connectivity preservation*. If there is a free path between two points, I can also find one based on the connectivity of the graph;

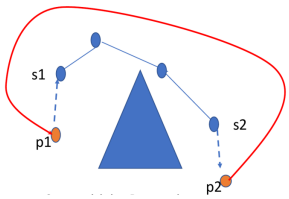


Figure 6: image representing this property.

- the solution found *grazes the obstacles*. In this way the paths that we obtain are the fastest ones;

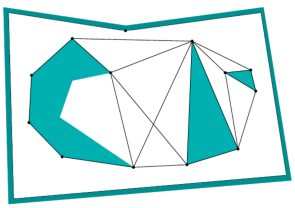


Figure 7: image representing the road-map created.

- it allows us to consider obstacles that are *not convex hulls*. We don't need to simplify complex polygon shapes because, when computing the road-map, it's like converting implicitly the polygons into convex hulls.

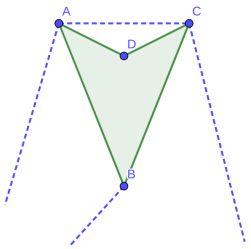


Figure 8: image representing a non convex obstacle. The path created doesn't consider the point D, but it goes directly from A to C.

Even though this road-map presents really good properties, there are some downsides, such as:

- the *computational cost is quite large*. In fact, the time complexity of the algorithm that builds the road-map is $O(|V|^3)$. This time complexity is not optimal but there are other algorithms able to construct a graph for a polygonal environment with holes in time $O(|V|^2)$. Despite its cost, we construct the map only once and is then shared among all the robots;

- the initial positions of the robots and the final destination are not included into the map. In order to do that, we use a built-in function that has a time complexity of $O(|V|)$. We have to use this three times, once per robot.

We decided not to use the *maximum clearance road-map* because, thanks to the implementation of the offset and our interpolation methodology, we are able to obtain safe trajectories. Moreover, we didn't use the *cell-decomposition algorithms* and alike because they don't create the fastest route as our method does.

2.4 Shortest-path discovery

Once the road-map has been computed and all the robots and exit gate have been connected to it, we need to find the *shortest-path* that connects all of them together.
To achieve this result, instead of using the classic **Dijkstra algorithm**, we used **A***.
The *A* Search algorithm* is one of the best and popular techniques used in path-finding and graph traversals and it allows us to overcome all the shortcomings found in *Dijkstra's algorithm*. For instance, there is no need of computing the optimal path from a node to all the others in the graph (and there can be many of them). Instead, we just need to obtain the path that goes from an initial point to the exit gate.
Moreover, the computational complexity of *A** is much better w.r.t. the counterpart (even if it's polynomial). In fact, the simplest implementation of *Dijkstra's algorithm* produces a time complexity of $O(|V|^2 + |E|)$ (and reaches $O((|E| + |V|)log|V|)$ if implemented with priority queues) while the worse case time complexity for *A** is $O(|E|)$ and the required auxiliary space needed in worst case is $O(|V|)$.
Although being the best path finding algorithm around, the results return by *A** heavily rely on its heuristics/approximations. In this case we use the *Euclidean distance* as the heuristic function.

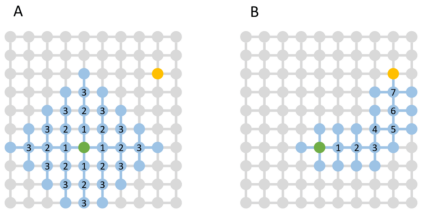


Figure 9: image representing the differences between **Dijkstra** *(A)* and **A*** *(B)*.

3 MOTION PLANNING

Now that we have all the necessary information to create a path, we move towards the so called motion planning area.
In this particular section we will implement all the motion strategies needed to create the *final trajectory* that each robot will follow in order to reach the final destination. More specifically, we will define our own interpolation methodology called **circle interpolation**.
Given the fact that our robot has some kinematic constraints (Dubins car kinematics) and the fact that the path obtained in the previous step is composed by a sequence of straight segments (as we can see in image 10), we need to interpolate every single one of them to make sure that the robot is able to follow the path correctly and smoothly.
In order to do it, we consider three different phases:

- *First trait* interpolation;
- *Middle traits* interpolation;
- *Last trait* interpolation.

We handled those tasks in two different ways. We used the *Markov-Dubins interpolating curves* for the first and last trait and a self-designed algorithm denoted as **circle interpolation** for the rest of the traits.

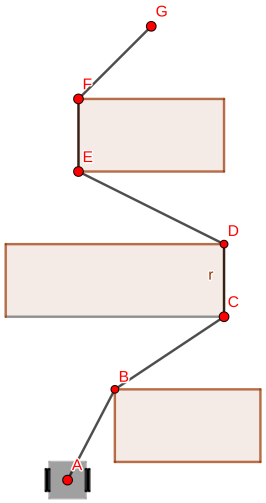


Figure 10: Image representing the path obtained after mission planning phase

3.1 First trait interpolation

We refer to the first trait as the segment connecting the starting point of the robot to the second node of the path. In example shown in image 10 it corresponds to the segment $\overline{AB}$.
In order to derive the interpolated trajectory for this trait, we simply refer to the *Markov-Dubins* method that uses:

- as *starting angle*, the initial orientation of the robot;
- as *final angle*, the angle of the second segment of the path (segment $\overline{BC}$ in figure above).

For the positions we use:

- the *initial position* of the robot as starting position;
- the position of the second node (node *B*) in the path as *final position*.

3.2 Circle interpolation

After defining the trajectory for the first-trait, we have to formalize the method that is able to find an arc that connects the *subsequent pairs of segments* that go from the initial trait to the final one (segments $\overline{BC}$, $\overline{CD}$, $\overline{DE}$ and $\overline{EF}$ in figure 10).

Given a pair of segments, we want to define a *circle* that is *tangent* to both of them. Once we found it, we can consider the two tangent points as the *entry* and *exit point* of an *arc* that belongs to the original circle. And so, by considering this new trait, we interpolate the path in such a way that the robot is able to move from one segment to the other smoothly.
Moreover, by defining the circle with radius equal to the *minimum curvature radius* of the robot, we will obtain an arc that is compatible with the *kinematics constraints* of the robot.
An example of trajectory created using this method is shown in image 11.

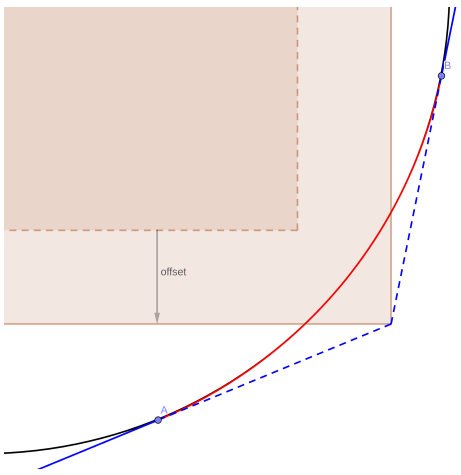


Figure 11: An example of circle interpolation

The red figure represents the offsetted obstacle and the blue line represents the path to interpolate; in order to obtain the right trajectory, we substitute the dotted blue line with the red arc

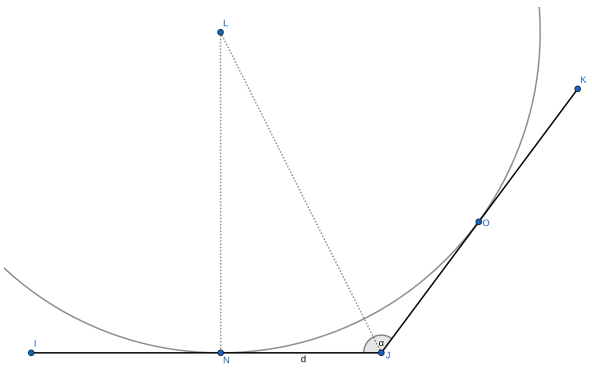
that connects the two points (entry and exit points) where the circle is tangent.
In order to find the entry and exit points in an efficient way, we used the following formula that computes the distance between this points and the *vertex* created by the conjunction of the segments:

$$d = r * \frac{1}{\tan(\frac{\alpha}{2})} \tag{1}$$

where α is the internal angle between the two segments and r is the radius of the circle (set to be equal to the minimum curvature radius of the robot). The derivation of this formula is shown in the following section 3.2.1.

3.2.1 Distance formula derivation

Given the following image:



Consider the segments $\overline{IJ}$ and $\overline{JK}$ which are the two sections of the path that we want to interpolate.
Consider also the segment $\overline{LN}$ of length r that starts from the center of the circle and arrives perpendicularly to $\overline{IJ}$.
The objective is to find the distance of the segment $\overline{NJ}$, denoted as d. If we consider the right-angled triangle composed by the points NJL and the angle formed by $\overline{LJI}$ (which is equal to $\alpha/2$), it's easy to notice that $\tan\left(\frac{\alpha}{2}\right) = \frac{\mathrm{r}}{d}$.
From this formula we can easily derive the distance formula (3).

3.3 Last trait interpolation

We refer to the last trait as the segment connecting the penultimate node of the path to the final one (segment $\overline{FG}$ in figure 10).
Also in this case, we use the *Markov-Dubins interpolation* to find the trajectory and we use:

- as *starting angle*, the angle of the penultimate segment of the path (segment $\overline{EF}$ in figure above 10);
- as *final angle*, the exit angle.

For the initial and final position we use:

- the position of the *penultimate node* of the path;
- the position of the *final node* of the path.

3.4 Analysis

We think that our circle interpolation algorithm is computationally more efficient w.r.t. the Multi-Point Markov-Dubins because the latter one computes all the possible Dubins curves, for then keeping only the optimal one. Moreover, by simply using Multi-Point Dubins, we don't have the guarantee of obtaining a collision-free trajectory. In fact, using Dubins we need to check if some collisions are occurring, which then translates into more computational burden to the entire system.
By utilizing our circle interpolation algorithm, we can generate a trajectory that is inherently free from collisions and this avoids the need for implementing further collision-checking methodologies.

However, in our implementation, the creation of the first and last trait of the trajectory doesn't guarantee a collision-free path. In fact, it could be that, in rare cases, the Dubins trajectories that we derive collide with the nearby obstacles.
We solved the problem for the last trait theoretically, with a technique explained in the future improvements section 7.

3.5 Adaptive Offsetting

The algorithm we implemented generates the shortest path by considering as nodes the vertices of the offsetted polygons. When we generate the trajectory with the *circle interpolation* technique, we interpolate the path by making it getting even closer to the obstacles, as it can be seen in the image 11. In order to avoid the collisions, it is not enough to apply an offset equal to half the size of the robot, we need something more.

Instead of using this simple approach, we tackled the problem with a more refined one that doesn't rely just on the size of the robot.
In fact, we applied a different offset size to each obstacle, obtaining the value by taking into consideration its shape as well as the robot's minimum curvature and size.

For a better explanation let's take into consideration the previous image 11.
We want to make sure that the robot travels around the corner without touching the obstacle. The offset is proportional to the angle of the corner.
In order to obtain a specific offset value for each obstacle, we derived the following formula:

$$d = r(1 - \sin\frac{\alpha}{2}) + \frac{h}{2}(\sin\frac{\alpha}{2}) \tag{2}$$

where r is the minimum curvature radius of the robot, h is the width of the robot and α is the smallest angle of the obstacle.
The smaller the angle, the greater the offset; thus, we choose the smallest angle of each obstacle

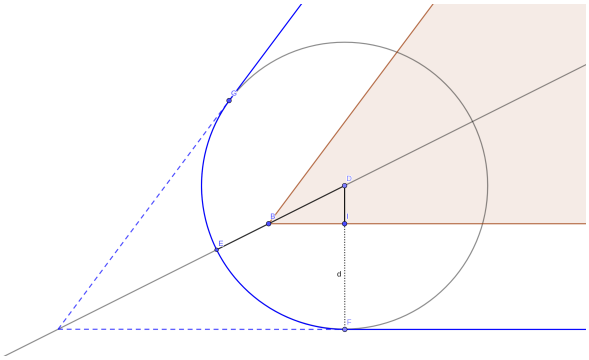
to compute a safe offset value. This allows to have a safe offset, also in the borderline case where the robot path is flanking the two sides of the obstacle.
Furthermore, the offset size should be at least half the size of the robot; therefore, for calculating the offset, we have to take the maximum between the result of the formula above and half of the robot's size. By doing so, we obtain the following formula:

$$d = Max\{r(1 - \sin\frac{\alpha}{2}) + \frac{h}{2}(\sin\frac{\alpha}{2}), \frac{h}{2}\} \tag{3}$$

In the following subsection we demonstrate how we derived this formula 2.

3.5.1 Adaptive offsetting formula demonstration

Let's put ourselves in the borderline case where the robot path is flanking the two sides of the offsetted obstacle.
Consider the following image:



We denote the offset size as d. The circle has a radius r equal to the minimum curvature of the robot.
The robot will follow the trajectory given by the arc $\overset{\frown}{FG}$.
As a result, both distances $\overline{BE}$ and $\overline{IF}$ (which represents the offset size) should be larger than half of the robot's size (denoted as h).

We will compute d as a function of r and $\frac{h}{2}$.
Consider the following equations:

$$\overline{ED} = r = \overline{EB} + \overline{BD} \quad \Rightarrow \quad \overline{EB} = r - \overline{BD} \tag{4}$$

$$\sin(\frac{\alpha}{2}) = \frac{\overline{DI}}{\overline{BD}} \quad \Rightarrow \quad \overline{BD} = \frac{\overline{DI}}{\sin(\frac{\alpha}{2})} \tag{5}$$

$$\overline{DF} = r = \overline{DI} + \overline{IF} = \overline{DI} + d \quad \Rightarrow \quad \overline{DI} = r - d \tag{6}$$

$$\sin(\frac{\alpha}{2}) = \frac{\overline{DI}}{\overline{BD}} \quad \Rightarrow \quad \overline{BD} = \frac{(r-d)}{\sin(\frac{\alpha}{2})} \tag{7}$$

We can use 7 to substitute $\overline{BD}$ in 4 to obtain:

$$\overline{EB} = r - \overline{BD} = r - \frac{(r-d)}{\sin(\frac{\alpha}{2})} = r(1 - \frac{1}{\sin(\frac{\alpha}{2})}) + \frac{d}{\sin(\frac{\alpha}{2})} \tag{8}$$

To obtain an equation in terms of d, we can start with the formula 6 and solve it for d:

$$\overline{EB} = r\left(1 - \frac{1}{\sin(\frac{\alpha}{2})}\right) + \frac{d}{\sin(\frac{\alpha}{2})} \Rightarrow \qquad d = \sin(\frac{\alpha}{2}) \cdot \overline{EB} - \sin(\frac{\alpha}{2}) \cdot r\left(1 - \frac{1}{\sin(\frac{\alpha}{2})}\right) \quad (9)$$

So the final equation is:

$$d = \overline{EB}\sin(\frac{\alpha}{2}) + r(1 - \sin(\frac{\alpha}{2}))$$

If $\overline{EB}$ is minimized to $\frac{h}{2}$, then the minimum value of d can be obtained by plugging $\overline{EB} = \frac{h}{2}$ into the equation we derived earlier:

$$d = \overline{EB}\sin(\frac{\alpha}{2}) + r(1 - \sin(\frac{\alpha}{2})) \ = \frac{h}{2}\sin(\frac{\alpha}{2}) + r(1 - \sin(\frac{\alpha}{2}))$$

Therefore, this formula gives the minimum possible value for d and given the fact that d cannot be less than $\frac{h}{2}$, we can modify the equation as follows:

$$d = Max\{r(1 - \sin\frac{\alpha}{2}) + \frac{h}{2}(\sin\frac{\alpha}{2}), \frac{h}{2}\}$$

4 ROBOT COORDINATION

At this point, we have all the necessary elements at our disposal to solve the **coordination task**. But before doing that, we have to take into consideration some important *intrinsic properties*. Thanks to the fact that the robot has a *constant velocity* and to the fact that the creation of the road-map considers the *shortest path design*, we are able to formalize the coordination task in a much more efficient and easy way.

The first intrinsic characteristic that we have to consider is that each pair of paths intersects *at most once* in a single point. In the most extreme case, this point coincides with the exit gate.

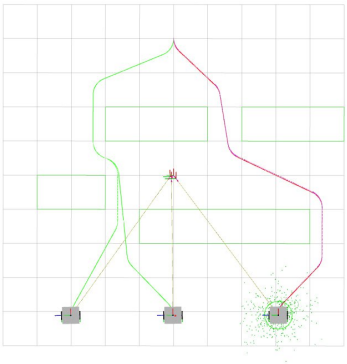


Figure 12: image representing the different paths and intersections.

The second and most important property that we have to consider is related to the conformation of the paths. Since we obtained the fastest path for every robot, to perform the coordination we can simply compute the *point of intersection* between the different paths and subsequently estimate the *travel time* towards that point. By doing so, we are sure that a robot is going to reach an intersection point in a fixed amount of time (constant velocity).
In order to locate the intersection point, we don't only consider the actual physical crossing-point between the paths but also the distance between them. More specifically, if the gap between a pair of paths is lower than half the size of a robot, we denote those two as being intersected.
Let's take into considerations this particular example.

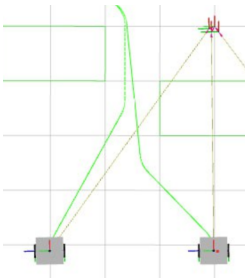


Figure 13: image representing two paths intersecting (denoted by the color green).

If the travel time needed to reach this point, for the two robots, differs by a value greater than a *minimum_delay* (10), the two robots will *never* hit each other even if *actuated at the same moment*.
On the other hand, if both of them reach the intersection point at the same instant, we can simply *delay* the actuation of one of them.
We are able to obtain the *optimal behaviour* by delaying the actuation of one of them by time defined in the following equation:

$$minimum\_delay = (size\_of\_robot + \rho)/constant\_velocity \tag{10}$$

In fact, this value represents the amount of time needed for a robot to travel a distance equal to its size and it assures that:

- the robots won't hit each other;
- the robots are relatively close to one another while traveling across the trajectory.

Thanks to this the robots are able to evacuate in the fastest way possible. We can also generalize this kind of behaviour for n robots.

The algorithm that we just explained can be formulated into this *three main steps*:

1. given a pair of paths, find the intersection point by taking also into consideration the distance ϵ. Do this for each possible pair of paths;
2. for each robot, find the length of the path-segment that goes from the starting point to the intersection point and divide it by the constant velocity. Thanks to this, we know how long it takes to arrive at that point;
3. compute the difference in the arrival time between the robots that intersect at the same point (we call this value Δ). If:
    - $\Delta >$ *minimum_delay*, send both of them at the same time;
    - $\Delta <=$ *minimum_delay*, apply the *minimum delay* to one of the two robots.

For this particular setting, we have to implement this strategy by taking into consideration 3 robots and so we have to examine all the possible combinations of intersections and Δ between the paths.

5 EXAMPLES

In this section, we present examples of simulations that demonstrate the effectiveness of our algorithm in computing trajectories and coordinating the actions of multiple robots.

5.1 Example 1

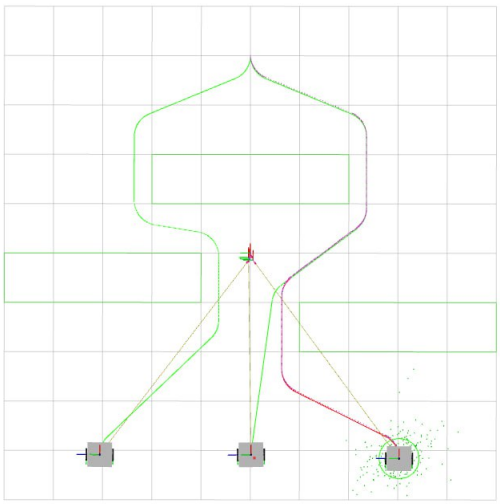


Figure 14: Example 1

In this particular scenario, there are three obstacles in the environment and a gate located at position $(0, 4)$ with an exit angle of $\pi/2$. The robots, in this setup, are labeled as robot 1, robot 2 and robot 3 from right to left. The algorithm initiates the movement of robot 1 and robot 2 concurrently while robot 3 is actuated after a delay of 2 seconds in order to avoid a collision with robot 1.

5.2 Example 2

In this particular scenario there are four obstacles in the environment. The robots and gate are located in the same positions as before. The algorithm initiates first the movement of robot 1, followed by robot 2 after a delay of 2 seconds. Finally, after another 2 seconds, robot 3 is actuated.

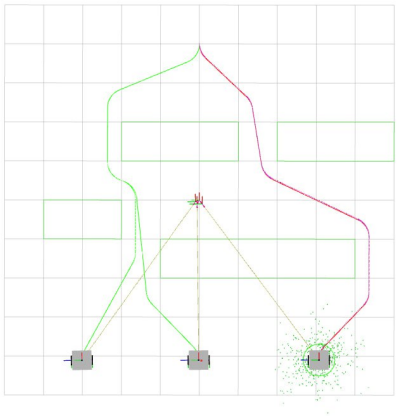


Figure 15: Example 2

5.3 Example 3

In this scenario, the environment is similar to that of Example 2, but there is a slight difference: the obstacle in the upper right corner is slightly larger. As a consequence, during the offset phase, this obstacle merges with the one on the left and the passage between them becomes inaccessible for the robots. The coordination algorithm initiates the robot's movements in the same order as Example 2, with robot 1 starting first, followed by robot 2 after a 2 seconds delay and finally robot 3 after another 2 seconds delay.

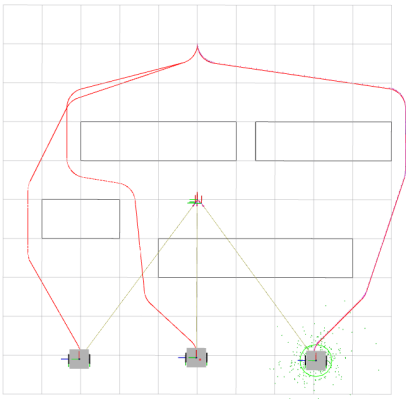


Figure 16: Example 3

6 CONCLUSIONS

In this project we tried to build the fastest possible trajectory for every robot. We were able to achieve such result by implementing the adaptive offset to the obstacles, by computing the road-map together with the shortest path and by performing the three-phases interpolation.
Moreover, we were able to coordinate the evacuation of the robots thanks to the computation of the intersection points between the paths and the computation of a minimum delay.
The final result is an algorithm able to evacuate the robots in a fast and safe way.

7 FUTURE IMPROVEMENTS

In this section we list some possible future improvements that can be implemented in our work.

7.1 Substitution of the first and last trait

In our method, as we saw in the previous chapters, we need to use the Dubins' curves to find a trajectory for the first and last trait.
However, there is an issue with it: there is the possibility for the robot to collide with the nearby obstacles while performing the maneuver. To solve this, let's consider separately the two cases.

Let's start by taking into consideration the last trait. We can imagine placing two circles (with radius equal to the minimum robot curvature) next to the exit point. These two circles are tangent to the center of the gate, which is also the only point in common that they have. Moreover, the center of these circles has the same y coordinate of the exit point. We divide this two circles into multiple sections and we consider the resulting polygons as new obstacles.
Thanks to this, our main algorithm finds the last point in the new obstacles and then follows the arc of one of the original circles to reach the exit point. By doing so, we are able to interpolate the last trait without using the Dubins' curves.

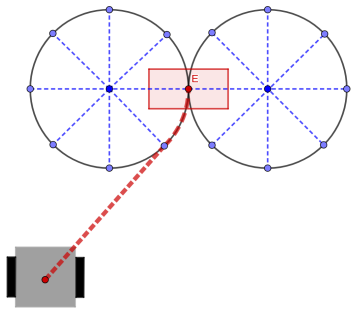


Figure 17: image representing the interpolation for the last trait.

A drawback of this procedure is that we cannot use it for the first trait because it doesn't guarantee the creation of a collision-free trait. Thus, it still remains an issue that can be solved in future works.