Robot Planning and its Application: Evader and Pursuer project

Luca De Menego, Edoardo Schioccola, Shuxin Zheng

January 12, 2022

Abstract

We present here the process followed in order to accomplish the fulfillment of the project of the course *Robot Planning and its Application*. Our solution has correctly created, with a complexity some collision-free Dubins shortest paths, and is used to move a robot in an arena with a given set of obstacles. In particular, we create two paths for two different robots, to emulate an Evader and Pursuer game in which the evader has to escape to one of the destination points, while the pursuer has to intercept it before it reaches the destination.

1 Overview of the project

The project involves an arena with either one or two robots and we are given their initial pose, some obstacles, and one or more possible destinations. The **first developed solution** simply calculates the collision-free shortest path for one robot to reach a certain destination. In order to do it, we:

- enlarge the obstacles twice, creating a "bigger" (enlarged by a factor equal to the radius of the circle circumscribed to the robot, which is represented as a triangle) and a "slightly-bigger" version of them. The biggest version is used to generate the shortest-path roadmap, while the other one is used for the collision detection phase. After this step, we can move our robot in the arena as if it is a simple point;
- generate the shortest-path roadmap using a visibility graph;
- find the shortest path between a source and a destination using the Dijkstra algorithm;
- find the multipoint Markov-Dubins shortest and a collision-free path between source and destination, passing through the nodes returned in the previous step (the shortest path);
- collisions are checked by verifying for each arc of the Dubins path if it intersects with any edge of the obstacles. The arc of a Dubins curve is an arc of a circle, so we applied a basic algorithm to check the intersection between a circle and a segment, investigating at the end if the intersection is really on the arc.

The **second developed solution** involves the actual goal of our project: the implementation of an evader and pursuer game. This is completed by the use of all the algorithms explained above, and a new one that enables the pursuer to intercept the evader, even if the latter one decides at some node to change destination stochastically. Details can be found in Section 3.

1.1 Main algorithms

The most important algorithms implemented (or used, in the case of the Clipper library) for the project's fulfillment, that will be explained in details in Section 2, are the following:

- **Visibility Graph generation**, in which we create the shortest path roadmap. The implemented algorithm follows the plane-sweep principle presented in Section 6.2.2 of the Motion planning book provided to us by the professor. In particular, the chapter 15 of Computational Geometry - Algorithms and Applications has been followed for the details of the visibility graph generation

implementation. To maintain the obstacle's edges sorted, we used a binary search tree called **OpenEdges**, that has been implemented following the open-source project pyvisgraph written in Python. The final complexity is $O(n^2 logn)$, where n is the number of vertices.

- **Dijkstra Shortest Path** which was implemented using C++ STL sets, that are self-balancing binary search trees. Final complexity: $O(ElogV)$, where E is the number of edges and V is the number of vertices of the graph.
- **Clipper library** which was used for polygon offsetting and to compute the "join" operation of obstacles.
- **Multipoint Markov-Dubins Problem**, which has been solved with an Iterative Dynamic Programming approach, as explained during class. The final complexity of the algorithm is $O(nk^2)$, where n is the number of points and k is the number of discretized angles.

2 In depth view of the project

2.1 Visibility Graph generation

When considering the problem of reaching a destination with a given starting position, there are different types of robots: some can move in straight lines, some other can rotate or even accelerate. However, in this section this has not been taken into account. Our goal is to find the Euclidean collision-free shortest path between two 2D positions, given an arena with a set of obstacles. It has been proven that the shortest path among a set of obstacles can be represented as a set of arcs of the visibility graph.

In order to compute the visibility graph, we need to find the pairs of vertices that can *see* each other, i.e. does the line segment connecting them intersects any edge of the obstacles? If not, we can create a new edge of our visibility graph. The naive algorithm that checks this for each pair costs $O(n^3)$: $O(n^2)$ because we have $(n \times (n-1))$ possible pairs of vertices, multiplied by $O(n)$ because we check the intersection with n total edges.

How do we reduce the complexity? As we can clearly see from figure 1, when checking whether a point can see another one we don't need to loop through all the obstacle's edges, if we create a special data structure. In the example, in fact, we would just need to check the edge $e1$.

In order to apply this idea, we need to do two things:

- consider the point in a certain order. In our case, we loop through them in counter-clockwise order.
- keep a binary search tree, that we'll call **OpenEdges**, to store the intersected edges in order. If implemented correctly, when considering the visible points from a certain point P, we will just need to consider the smallest edge of the OpenEdges structure, that will always be the leftmost leaf.

In details, this was implemented in the following way:

- let us consider that we want to find the visible vertices from a fixed point P;
- first we want to sort all the vertices by counter-clockwise angle, where the calculated angle is the one that each segment from P to the chosen vertex makes with the positive x-axis. If the angle is the same for two given vertices, we consider as *smaller* the ones that are closer to P;
- now consider the half line t starting from P and pointing to the positive x direction;
- this line moves in a counter-clockwise way, considering one vertex after the other;
- when reaching a certain vertex Q, we consider all the obstacle's edges incident on Q. If an edge lies on the counter-clockwise side of t, insert it into *OpenEdges*, otherwise delete it;
- during this process, we decide if each point is visible or not by checking if the half-line t intersects the smallest *OpenEdge*.

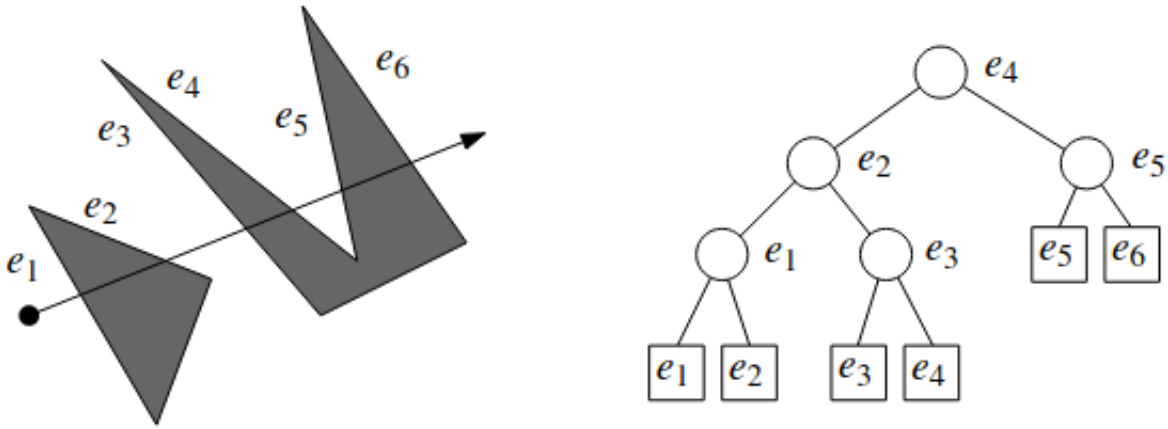


Figure 1: The use of the *OpenEdges* structure

This process seems pretty straightforward, but there are a lot of **corner cases** we need to consider when deciding whether a point is visible or not from P, and all of them happen when the half-line t contains multiple vertices. We provide in the figure 2 some examples. The idea to solve these problems is the following:

- if w_{i-1} is not visible, w_i is obviously not visible too;
- if w_{i-1} is visible, then w_i is not visible if at least one of the two following conditions hold:
  - the segment w_{i-1} - w_i lies in an obstacle;
  - the segment w_{i-1} - w_i intersects an edge in *OpenEdges*.

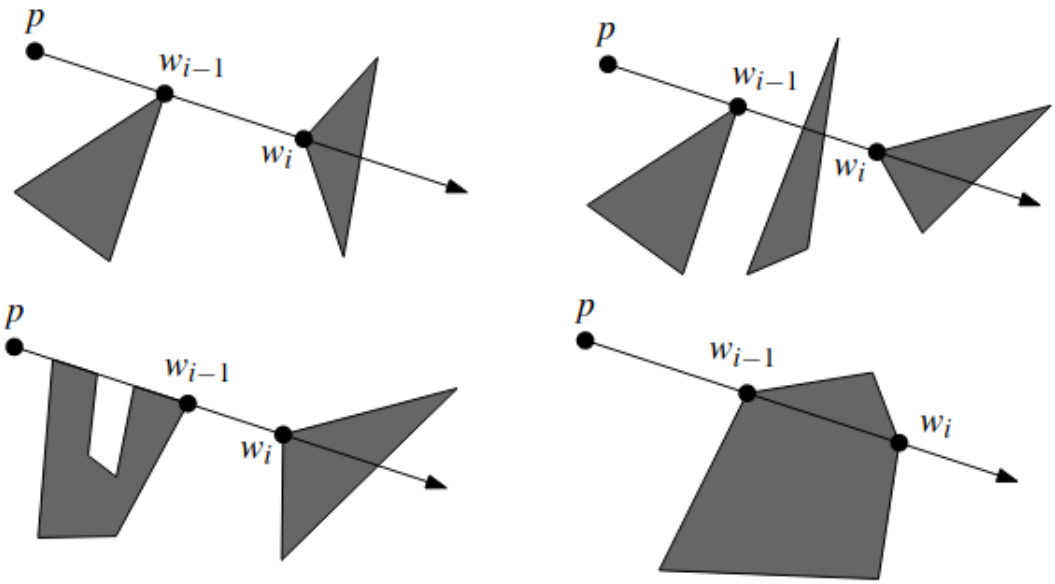


Figure 2: Corner cases of the presented algorithm when checking if a point is visible from p

What about the complexity? When considering the visible vertices from a fixed point, the time spent is dominated by the first sorting of the vertices in counter-clockwise order, which is $O(nlogn)$. When repeating this process for n points, we clearly get an overall complexity of $O(n^2logn)$.

2.2 Clipper Library

Clipper is a library used for line & polygon clipping, intersection, union, exclusive-or, and line or polygon offsetting. We used clipper mainly to enlarge polygons that are present in the arena in order to keep a "safe distance" from obstacles that could cause a collision. Our newly implemented functions are collected in the file "clipper-addons.cpp" in which we can find:

- **enlarge**: which applies an offset to a polygon given as input and returns the enlarged version of the polygon. The *Join Type* chosen for this phase if *JTMiter*, that limits potential spikes that may happen when offsetting edges that join at very acute angles.
- **printSolution**: displays via OpenCV the solution of a clipper polygon offsetting.
- **intersect**: verifies if two clipper polygons intersect, returns true if they do, false otherwise.
- **enlargeObstaclesWithTwoOffsets**: given a vector of obstacles and an offset, it creates a "bigger" version and a "slightly bigger" one, then returns both. The slightly bigger version of the obstacles is used for the generation of the shortest path roadmap because the use of this kind of roadmap is risky, since our robot should move very close to the vertices of the obstacles. Using some slightly bigger obstacles make the path much safer, as we can see from the examples in section 2.4. After having constructed the map, we can finally use the smaller obstacles for the collision detection phase.
- **enlargeAndJoinObstacles**: verifies if, after being enlarged, two polygons intersect. If they do, it joins them creating a single polygon.

Another important element we took into account was how to find the optimal "distance" (or enlargement) to keep between the robot(both pursuer and evader) and the obstacles. The optimal value was found first by defining, via the calculus of the distances from the center to all the vertices of the triangle what type of triangle it was so that we could find the perfect circle to in capsule the triangle. The distance that was chosen was the "largest", as by choosing that we would have been able to avoid every possible problem that could arise by rotating the robot.

2.3 Multipoint Markov-Dubins problem

Given an initial (x_0, y_0, th_0) and a final (x_f, y_f, th_f) configuration, with a bound on the maximum curvature k_{max} and a constraint on the direction of motion of a vehicle, the **Dubins Curve** is the optimal curve satisfying these constraints. It is composed by three arcs, that can turn left, right, or go straight, and it has been proved that only the following combinations are possible optimal solutions: LSL, LSR, RSL, RSR, LRL, RLR. Obviously, the other possible solutions composed by less than three arcs are possible, but they are already inherently present in the first 6 combinations.

The actual solutions to check each configuration can be found in the project's source code on GitHub. The value of k_{max}, on the other hand, was found iteratively by checking the motion of the robot. In particular, we have found out, after some tests on specially crafted paths, that the highest value of k for which the robot is still able to accurately follow a path is 20.

Now that we know what a *Dubins Curve* is and how to solve a problem in which we are simply given an initial and a final configuration, how do we **generalize** the problem for finding the shortest path given a set of intermediate nodes we have to pass through and a starting configuration (x_0, y_0, th_0)?

This is called the **Multipoint Markov-Dubins Problem**. In our case we can define it in the following way:

- we are given a sequence of assigned planar points P_0,P_1, ..., P_n and a starting angle th_i;
- we consider a fixed set of k possible joining angles (e.g. $[0, \pi/2, \pi, 3\pi/2]$)
- we want to find the shortest Markov-Dubins path that starts from P_0 with a fixed angle th_0 and passes through the points P_1, ..., P_n.

The brute-force approach of trying all cases for each pair of nodes is clearly not feasible, due to their exponential number.

That's why an **iterative dynamic programming approach** has been used, where we define a function $D_j(\theta_j, \theta_{j+1})$ as the length of the optimal solution of the two points Markov-Dubins problem, and we want to find the optimal angles $[\theta_0, \theta_1, ..., \theta_n]$ that minimize the total length L of the path.

In order to solve it, we defined a function $L(j, \theta_j)$ that gives the length of the solution of the sub-problem from point P_j with angle θ_j onwards. The solution can be described recursively in the following way:

$$L(j, \theta_j) = \begin{cases} min_{\theta_n} D_{n-1}(\theta_{n-1}, \theta_n) & j = n-1 \\ min_{\theta_{j+1}} D_j(\theta_j, \theta_{j+1}) + L(j+1, \theta_{j+1}) & otherwise \end{cases} \quad (1)$$

We need to compute $O(k^2)$ two points Markov-Dubins problems, and then do a for loop in which we follow the upper equation, from $j = n-2$ to $j = 0$. This means the overall complexity is $O(nk^2)$.

In order to **reconstruct the final solution**, during the implementation a matrix S has been used, that contains which angle each configuration of L wants to finish with. In this way, at the end, when we have the first minimizing angle that corresponds to the shortest path, we can iteratively follow the matrix S constructing our final result: the array of minimizing angles $[\theta_0, \theta_1, ..., \theta_n]$.

There is only **one problem remaining**: the solution explained above is actually a solution for another type of Multipoint Markov-Dubins problem, in which we don't have a fixed initial angle, but a fixed ending one. How do we convert a solution of this problem to a solution of our problem?

What we want to do is to find a reduction from our problem to the problem we have just solved. The reduction is the following: we convert the input I of our problem to a new input I', such that:

- the new array of points is in reversed order with respect to the original one;
- the source angle θ become the destination angle $(\theta + \pi)$.

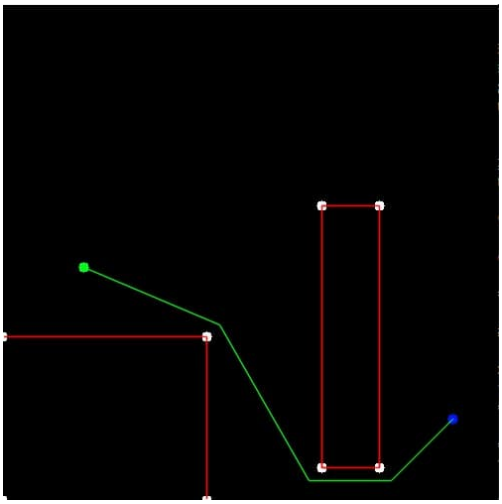
With this new input, we will have a solution that will have to be finally converted back to a solution of our problem. Even in this case, the transformation is really simple:

- the resulting array of optimizing angles is reversed;
- to each resulting angle we add π.

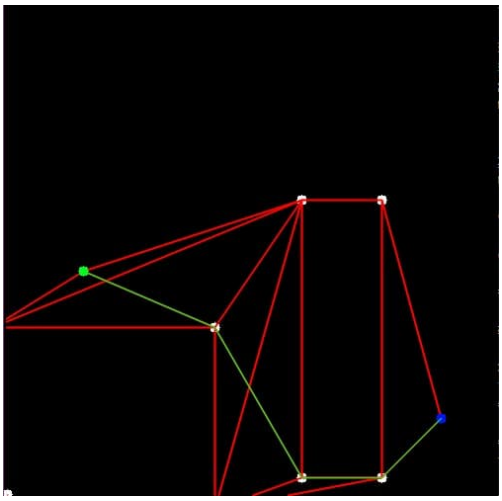
Now, having the optimizing angles, we can just solve for each pair of points the two points shortest path Markov-Dubins problem, and return the array of resulting *Dubins Curves*.

2.4 Examples of our results

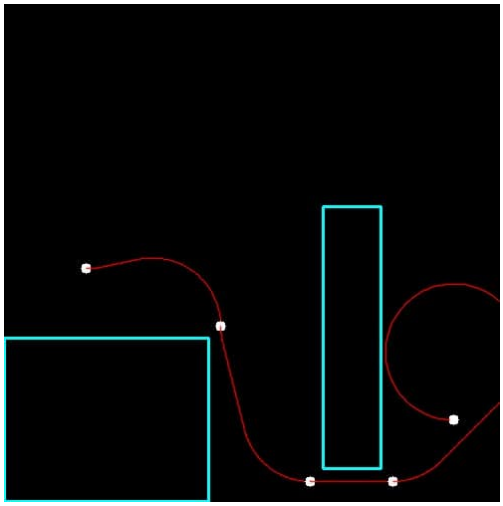
Here we present some examples of our results, generated with openCV or taken as screenshots from RVIZ.



(a) Shortest Path

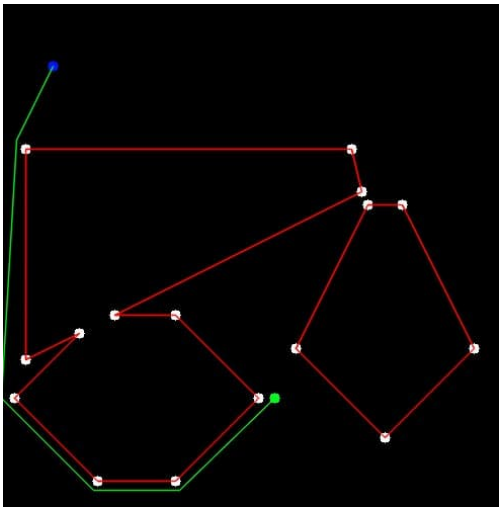


(b) Visibility Graph

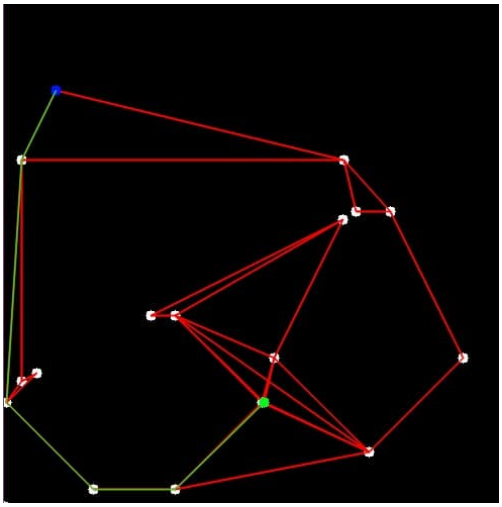


(c) Dubins Path

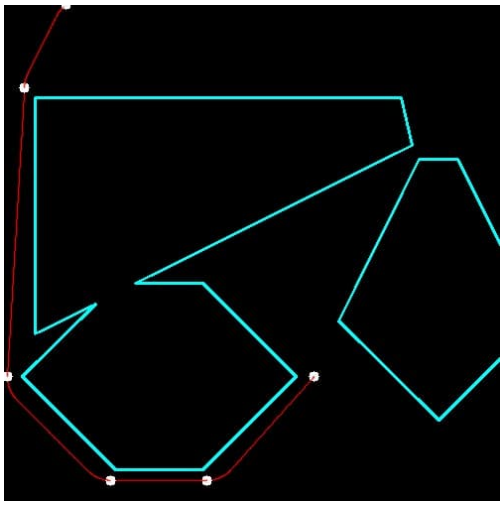
Figure 3: All the visualization modalities for a path



(a) Shortest Path



(b) Visibility Graph



(c) Dubins Path

Figure 4: All the visualization modalities for a path

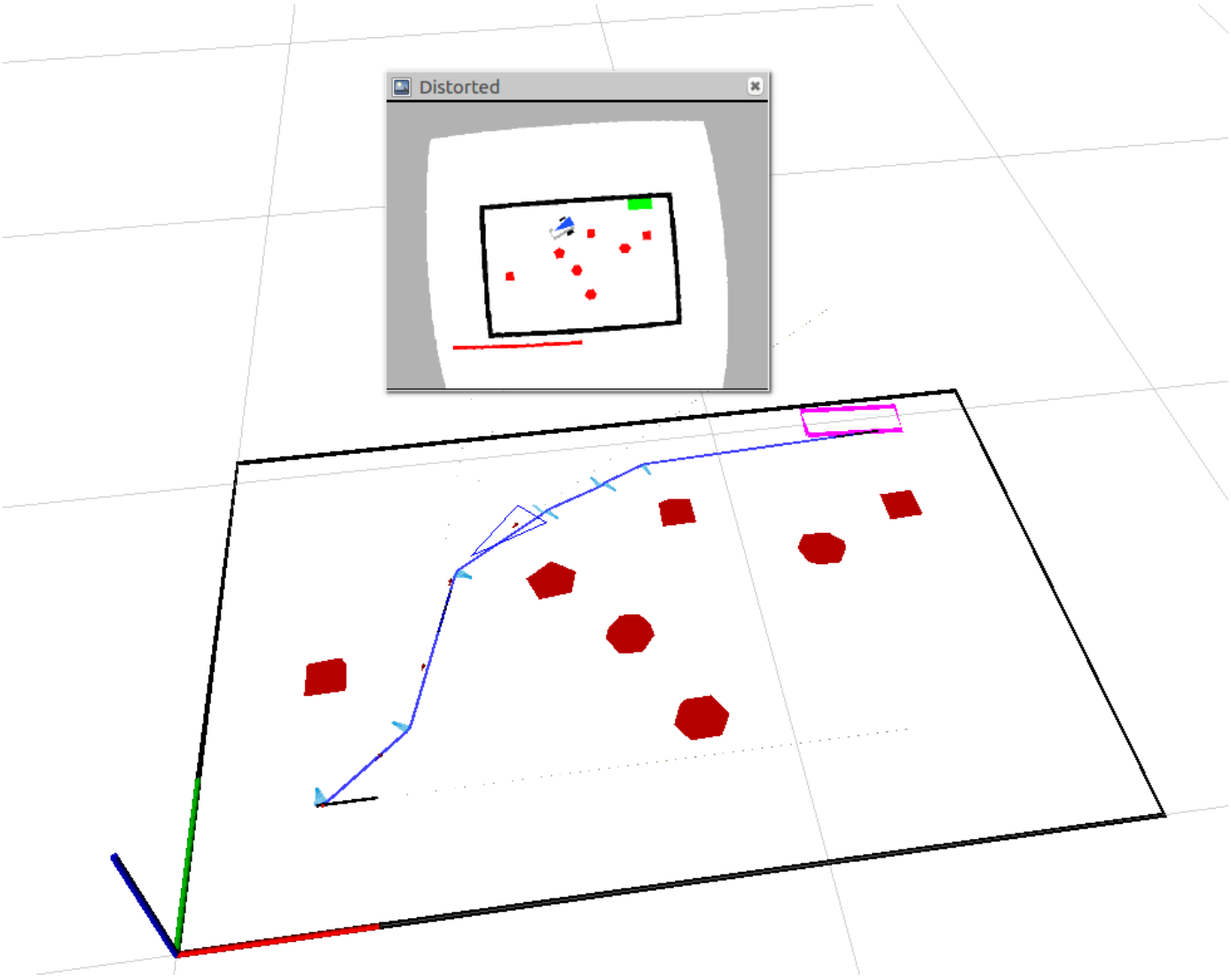


Figure 5: Example of multipoint Markov-Dubins path, seen from RVIZ

3 Pursuer and Evader game

We divided this part of our project in two, based on the type of arena we are dealing with. In particular, if there is one only exit available, the pursuer will be much more facilitated in its work. This is because, while when there are more exists the evader can decide where to go, and even change its mind at runtime, if there is one only exit available the pursuer will already know the evader's plan.

3.1 One destination available

Since the roadmap generated during initialization is the same for evader and pursuer, there will only be one possible shortest path for the evader.

What about the pursuer? How does it intercept the evader? The developed algorithm takes as input the shortest path of the evader, and looping through all nodes of it, from the second one to the destination:

- calculates the pursuer's shortest path to reach the chosen node;
- checks if the length of the calculated shortest path is less than the length of the evader's shortest path from its source to the chosen node:
  - if the pursuer's shortest path is longer than the one of the evader, continue looping choosing the next node;
  - otherwise we have a possible point of interception. So we calculate the Multipoint Markov-Dubins from the pursuer's source to the chosen node:
    - if the path is found and its size is less than the evader's one, we have finally found the node in which the pursuer can intercept the evader;
    - otherwise we continue looping choosing the next node.

If after this loop we do not have a valid solution, it means there is no way for the pursuer to catch the evader. In figure 6 it is possible to see an example of a path generated for the pursuer following the explained algorithm.

3.2 Multiple destinations available

The **evader now has a new feature**: it can **stochastically change its destination goal** at runtime. We defined a fixed probability p of changing it, that is evaluated each time the evader reaches one of the nodes of the graph. However, if the evader finds out that there is no way to reach the newly decided destination, it will stay on the previously chosen path. This means **it won't get stuck**, as long as there is a path available from its initial position to one destination.

How does the **pursuer** react to these sudden changes? The algorithm described in Subsection 3.1 is still used, but refined and repeated each time the evader reaches a new node of the graph. This is why the complexity, in this case, becomes much higher: the pursuer now has to recalculate its path at each new node, while trying to understand where the evader wants to move. In order to simulate the fact that the pursuer should not be able to predict the future, we allow it to change its path accordingly to the evader's new position only after:

- it reaches the next node (because of the synchronous assumption of the project).
- the evader has reached a new node (this is done by comparing the length of the Markov-Dubins sub-paths);

When these two conditions are verified, the pursuer:

- just moves closer to the destination that is closer to the evader's position if it's the first iteration (at first the pursuer doesn't have any clue on the evader's future decisions);
- otherwise it tries to understand whether the evader is going towards the first or the second destination;

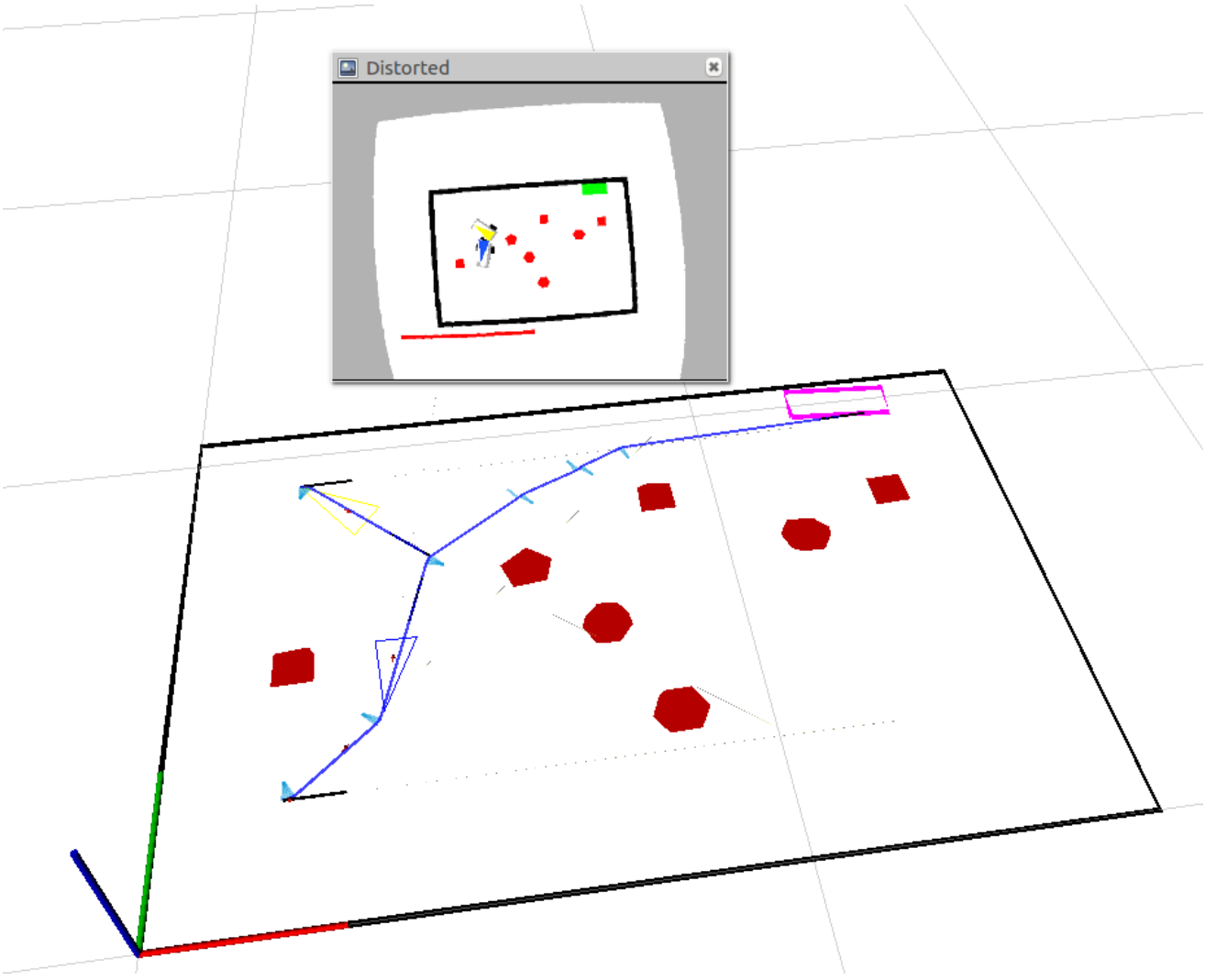


Figure 6: Example of a pursuer correctly intercepting an evader, seen from RVIZ. The pursuer is the yellow robot, while the evader is the blue one

- if it's too difficult to understand where the evader is going, the pursuer simply moves towards the evader's current position;
- otherwise, it applies the algorithm explained in Subsection 3.1.

It is possible to see an example of a path generated for the pursuer in the case of multiple destinations and an evader changing its mind at run-time in figure 7.

We take this algorithm out of the following consideration: since the pursuer can only take actions after the evader's ones, the smartest choice at the first iteration for the pursuer may seem to move somewhere between the two destinations. In this case, however, if the evader initially decides to goes directly to its closest destination, the pursuer may not be able anymore to catch it. Instead, making the pursuer first move towards the destination that is closer to the evader's position:

- if the evader goes to that destination too, the probability of catching it for the pursuer is maximised;
- if the evader goes to the other destination, since it was further the pursuer may still be able to catch it.

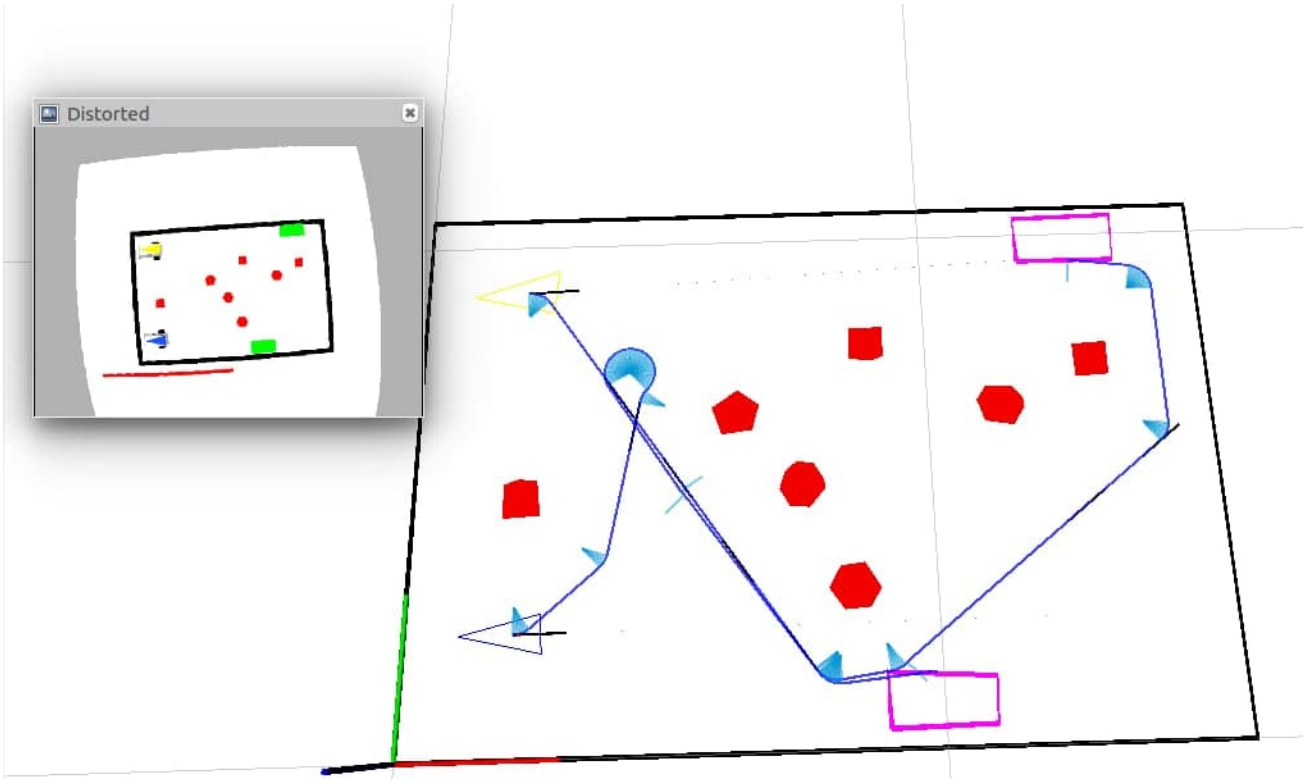


Figure 7: Example of a pursuer correctly intercepting an evader, seen from RVIZ. The evader changed its mind at runtime about the final destination. In this case, it was not a smart choice. The pursuer is the yellow robot, while the evader is the blue one

4 Problems and Limitations

A fundamental point to cover in our project is the possible problems alongside the limitations that may occur:

- following the example displayed in 8, we have that by following the approach described in the previous paragraphs, the pursuer eventually reaches a state in which he got stuck and is no longer able to decide a consequent path thus being unable to proceed. This happens because the

robot is not able to rotate on itself, and it is too close to the borders (that have been previously enlarged, as all the other polygons). This makes it impossible for the pursuer to move. This can obviously happen whenever we have an exit near both the pursuer and the evader. The problem could be solved either by modelling the robot in a way that allows it to rotate on itself for a more free of bounds movement, or by adding more intermediate nodes in the roadmap. But the first one may also be a limitation of real world robots, while the latter will lead to an important increase in complexity.

- Another limitation comes from the synchronous assumption of the project: the robots can only change their paths after having reached a new node of the roadmap. This means the pursuer will never be able to react effectively to sudden changes in the path of the evader.

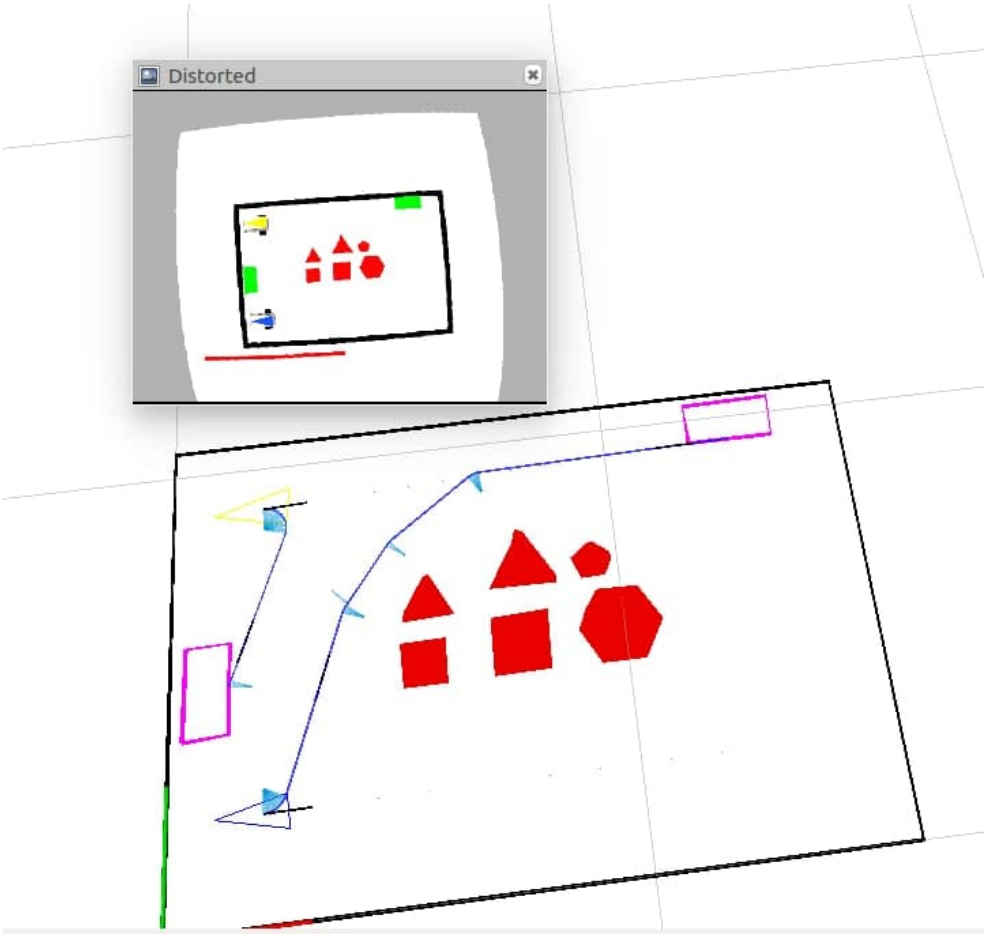


Figure 8: Example of a possible problem that could arise when using our approach