University of Trento

Robot Planning and its application - 145874 - Prof. Palopoli

Pursuit-evasion project report

Simone Peroni, ID 230406
simone.peroni@studenti.unitn.it

Mario Tilocca, ID 230394
mario.tilocca@studenti.unitn.it

Source code: https://github.com/SimonePeroni/AppliedRoboticsStudentInterface

Code documentation: https://mariotilocca98.github.io/document.github.io/

Trento, January 17, 2022

Contents

1 Introduction

In this report we present the main project choices in the development of a robot pursuer-evader game. For a look at the technical implementation, refer to the source code and the documentation linked in the front page of this report.

The goal of the project was to build a road-map for safe movement inside an obstacle arena and develop a strategy to plan the motion of two robots in an asynchronous pursuit-evasion scenario. The first section of this report explains how the robots were treated as oriented points in the two-dimensional plane. In the second section, we show how the road-maps are computed, including why a visibility graph approach was preferred over a maximum clearance one. The navigation system employed to find the shortest paths in the pre-computed road-map, based on the common *Dijkstra* algorithm, is presented in the third section. The fourth section describes the implementation of the solution to the *Markov-Dubins* problem. The last section presents the strategy employed to catch the evader, whose moves are randomly chosen. In the appendices, additional information on utility functions and some results obtained from the simulator can be found.

2 Robots as oriented points: obstacle inflation

Since the whole game takes place in a two-dimensional arena, it is obvious that the height of the robots will be ignored throughout the analysis. On the other hand, the presence of obstacles - although the walls of the arena would already be a sufficient motivation - forces us to consider the encumbrance of the robots in the 2D space. Treating the robots as polygons, however, would make the geometrical steps of the process, such as collision detection, quite complex. For this reason, the simplest option is to consider the robots as points with an orientation and to inflate the dimensions of the obstacles to account for the actual size of the robot, granting collision avoidance for every trajectory not intersecting the inflated set of obstacles.

Obstacle inflation is the first operation that takes place even before building the road-map. Following the suggestions received during the course, we used the *Clipper* library to perform this task. This is the only external library used in the final solution. This library allows to perform several operations with polygons, including the offset operation.

The inflation of polygons happens separately for the obstacles and for the borders of the arena for two reasons. First, obstacles require outer offsetting as the resulting polygons will be bigger than the original obstacles, while the borders of the arena require inner offsetting for the resulting polygon will be smaller than the original borders. Second, not all collision avoidance operations require to avoid both the obstacles and the borders of the arena. This is the case for the last segment of path leading a robot through an exit gate, that requires traversing the arena borders.

3 Road-Map

3.1 A heavy pre-computing approach

The most important choice of the whole project was to perform as much computation as possible before running the actual game. The reason behind this was to have a performant solution in a real-time scenario, where a robot could not take more time computing the next move than the time between two events that trigger a replanning task. Following this idea, we decided to precompute all possible path pieces in the roadmap building process. This is a sensible approach given the limited dimensions and complexity of the arena. In bigger and more complex arenas, most of the paths would be pre-computed uselessly, as every segment would have a very small probability of being traversed in the actual running task. Computing multipoint Markov-Dubins curves at real-time, however, would require real-time collision checking and it would be hard to re-compute alternative paths in case a collision on the theoretical shortest trajectory was detected. Therefore, a small arena and a simpler planning scenario motivate the choice of pre-computing the best Dubins curves connecting a discrete set of poses in the arena.

The road-map was implemented with a two-level graph representation. The base directed graph contains a set of interconnected xy-position nodes. This part is created arbitrarily, for example from a voronoi diagram or from a visibility graph, or even from sampling-based methods. The chosen approach for this project is explained in the following subsection. The more complex directed graph of the road-map is then computed

from the base graph trying to find feasible (i.e. existing and not colliding with obstacles) Dubins curves connecting a finite set of poses for each node to each pose of every node connected to the former one in the base graph. After this graph is computed, planning is easily performed as every building block is already stored in memory.

3.2 Maximum Clearance Graph & Visibility Graph

Given the heavy pre-computing approach, sampling-based methods were soon discarded, for their very high number of nodes required to exhaustively describe the arena. Among the geometrical methods, our first choice was to create a maximum clearance graph. Such graph is obtained by extending the Voronoi diagram concept to taking segments as input instead of points. The obtained graph has a simple topology, with a limited number of points and very few connections, as each node is connected to two or three others at maximum. Such case seems optimal for the pre-computing idea, and at the same time allows the robots to move safely at the maximum distance from the obstacles, although the paths would never be the optimal shortest ones. For this approach, another external library, *Boost.Polygon*, was used to compute the Voronoi diagram of inflated obstacles. This implementation can be found in the *old_implementation* branch of the source code, but was removed in the final solution. In fact, the simplicity of the graph, and more specifically the low number of connections for each node, showed problems very soon. Some edges of the graph were so short that weird solutions of the Dubins problem would appear at those places, for the curve radius due to the maximum curvature was disproportionate to the length of the edge. These resulting solutions would see the robots spinning around in a circle before reaching the other node although it was clear that better paths could have been taken.

The first attempt to solve this problem saw the creation of bypass connections around short edges, increasing the complexity of the graph but giving an alternative path to those edges. As this step came to require more passes and the implementation began to become more and more convoluted, the benefits of using a maximum clearance road-map soon faded away. Since we noticed that the road-map building times were not too bad, we decided to try using visibility graphs, opposite to maximum clearance graphs under every perspective. Visibility graphs, in fact, have a high complexity since they create the maximum number of feasible connections starting from each node, and move very close to the obstacles, allowing to follow the actual shortest path. A comparison between the two approaches is shown in Fig. 1.

Since visibility graphs have always showed good results since their introduction in the project, they were used in the final implementation. Of course, the very high number of edges in the graph made road-map building the bottleneck of computing time in the whole process. In the end, we sacrificed computing time for the sake of a nice-looking solution of the planning problem.

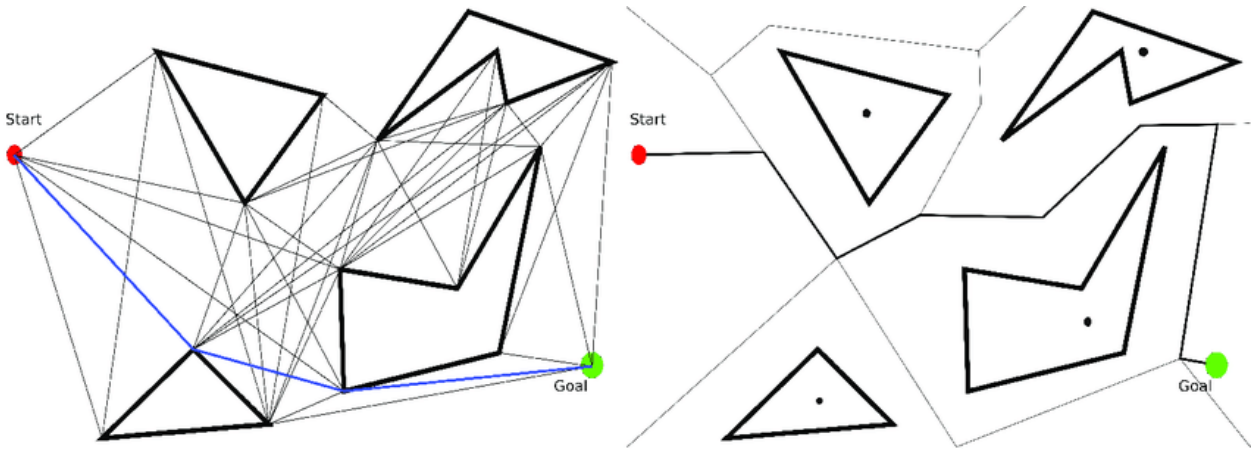

Figure 1: Maximum clearance graph and visibility graph comparison [1]

4 Navigation: Dijkstra algorithm

For the navigation part, two popular algorithms were initially taken into consideration to find the shortest path between two nodes of the road-map: Dijkstra and A*. The main advantage of using A* over Dijkstra is that A* finds the shortest path by exploring a smaller portion of the graph compared to Dijkstra. However, our key choice of pre-computing as much as possible has an influence here as well. In this case, once the road-map is ready, we want to pre-compute all the escape routes through the gates, therefore requiring a full exploration of the graph. For this reason, Dijkstra is the algorithm we chose, as A* would not have given any advantage for this purpose.

A major drawback of the Dijkstra algorithm is its $O(V^2)$ complexity, where V is the number of nodes in the graph. Thus, depending on the map configuration, there might be a high number of nodes leading to long runtimes. However, there is the possibility to reduce the complexity of the Dijkstra algorithm by introducing a minimum priority queue, meaning that the elements with lower values are given a higher priority. In this way the time complexity becomes $O(V + E \cdot log(V))$, where E is the number of edges in the graph.

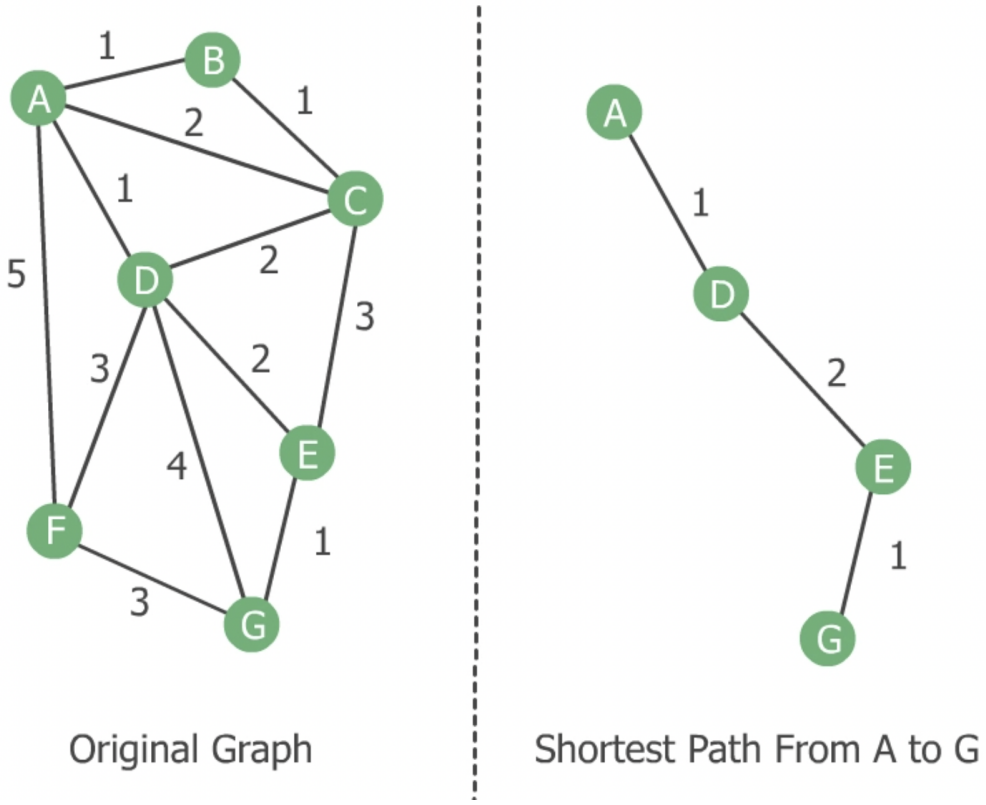

Figure 2: shortest path finding example [2]

Fig.[2] shows an example of a shortest path finding problem, where in our case the edge weights correspond to the length of the Dubins curves.

Here we present the pseudo-code followed in our implementation of the algorithm:

Algorithm 1 Dijkstra algorithm pseudo code

```
1: function DIJKSTRA(G, S)
2:     for each vertex V in G do
3:         distance[V] ← infinite
4:         previous[V] ← NULL
5:         if V! = S then
6:             add V to Priority Queue Q
7:             distance[S] ← 0
8:         end if
9:     end for
10:    while Q IS NOT EMPTY do
11:        U ← Extract MIN from Q
12:        for each unvisited neighbour V of U do
13:            tempDistance ← distance[U] + edge_weight(U, V)
14:            if tempDistance < distance[V] then
15:                distance[V] ← tempDistance
16:                previous[V] ← U
17:            end if
18:        end for
19:    end while
20:    return distance[], previous[]
```

In our implementation, the returned *distance* and *previous* values for each pose in the road-map are stored in what we call "navigation map", which is simply a road-map that was pre-computed for navigation. Instead of storing the previous node, however, we store a pointer to the Dubins curve connecting the two poses, making it easy to reconstruct the best path to another node. The algorithm was adapted to work also in a backward direction, that is starting the exploration from the goal by running every connection backward and saving the distance it takes from every other pose to reach the goal. This is the case used for the pre-computed navigation maps to the gates. Forward navigation maps are used by the pursuer at runtime and get recomputed in real-time each time the evader makes a new move. Using the A* algorithm in this case would bring some benefits as a full exploration of the map is not necessary anymore, but the current approach showed good computing times.

5 Dubins problem

In the given simulator setting, the robots are nonholomonic and can only travel forward, therefore their kinematic model can be described with the following equations in a Euclidean 2D space:

$$\begin{aligned} \dot{x} &= v \cdot cos(\phi) \\ \dot{y} &= v \cdot sin(\phi) \\ \dot{\phi} &= \omega \end{aligned} \tag{1}$$

where v is the constant velocity at which the robot travel, ϕ represents the heading, ω the turn rate and $\dot{x}$ and $\dot{y}$ represents the lateral and longitudinal velocities of the robot. With this assumptions in mind, the task is to find the shortest curve which connects two nodes given the initial and final angles of the robot. The turning rate is bounded to the chosen maximum curvature, thus the Dubins path joining two nodes is always composed by straight lines and arcs of the maximum curvature.
Approaching the problem from a mathematical point of view, it can be defined as a minimization problem over time which has the goal of finding the curve with minimum length that joins the two points.
Therefore the task is given the initial and final configuration of the robot and a maximum curvature k_{max} we need to compute all feasible combinations and choose the optimal one. Among the total 15 possible combinations, only the ones where 3 arcs are employed to construct the Dubins curve are taken into account.

Knowing that **L** represents a left turn, **R** a right turn and **S** a straight line; the possible candidates for the optimal solution are shrunk to 6: *LSL*, *LSR*, *RSL*, *RSL*, *RSR*, *LRL*, *RLR*.
In order to determine which one is the optimal curve, all 6 solutions need to be computed and then it is evaluated which one is the shortest as shown in Fig.[3].

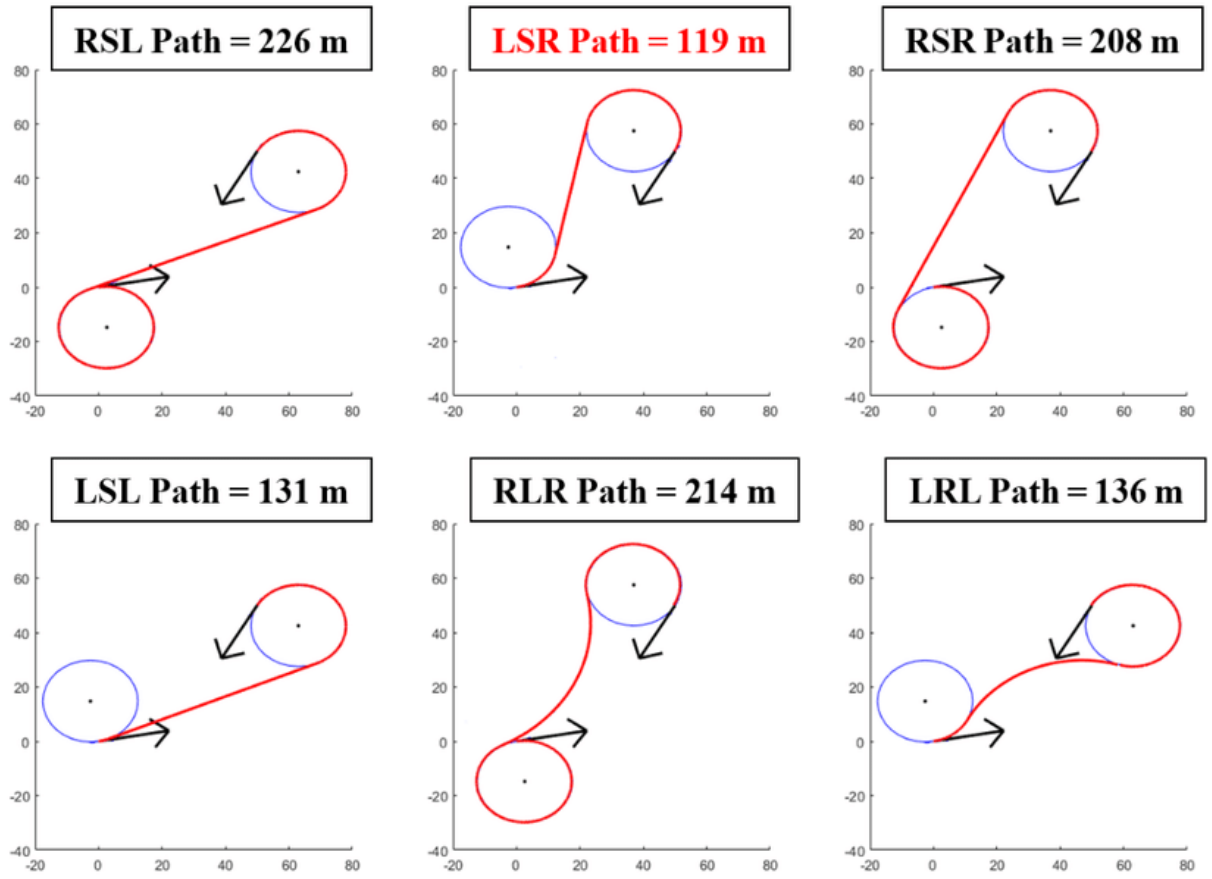

Figure 3: Dubins curve optimal solution example [3]

From a mathematical point of view in order to calculate which of the 6 above mentioned primitives is optimal, there is the need to simplify the closed form of the problem for facilitating the calculation. This is achieved by scaling the general setting to the standard setting, therefore a specific function with this purpose was built, together with one to scale from the standard form back to the original problem. Moreover, before a candidate solution can be accepted a crucial step is that all 3 cardinal equations are satisfied; these equations concern the line segment and positive curvature for left turning and negative curvature for right turning. Only if all 3 equations are satisfied a solution can be accepted. In order to construct the cardinal functions a special auxiliary function *sinc* needs to be previously implemented in a numerically stable way.

6 Pursuer-Evader strategy

As mentioned in the beginning, the mission is structured with two different robots playing different roles. The evader aims at reaching one of the gates and escape from the arena. The pursuer, as the name suggests, has the objective of catching the evader before it is able to exit the arena. The game ends when one of the two robots achieves its goal, but in our implementation the pursuer will follow the evader to the gate if it manages to escape.

In our game, the evader is not aware of the pursuer presence in the arena and whenever it reaches a new

node it randomly chooses one of the gates and plans the best path to it. In case the chosen gate is different from the one selected at the previous node a replanning of the shortest path leading to that occurs.

Conversely the pursuer is always aware of the evader's position and the goal it is heading to. This strong assumption has been made in order to facilitate the asynchronous planning of the pursuer-evader strategy. This simplification is justified, as the team believes that based on the short-term past moves of the evader it would be possible to estimate and predict what gate the evader is heading to. However, in this implementation, in order to create a simpler deterministic scenario at planning time the current goal is assumed to be known, as such estimation would be out of the scope of this course.

In order to run the game, a few ingredients are required: the pre-computed navigation maps for each gate, an empty navigation map attached to the road-map that will be re-computed multiple times by the pursuer as it moves, and the starting configurations of both robots. At the end of the game, the full paths followed by the robots from start to end are obtained.

The behaviour of the evader is trivial and does not require further explanation. The pursuer, however, follows a specific strategy to catch the evader. Knowing the gate the evader is heading to and that it takes the shortest path to it, the pursuer can compute the expected path of the evader. At this point, the pursuer will simply head to the first node along this path that can be reached before the evader, catching it at the first opportunity.

It is not granted that the pursuer will always catch the evader. Some maps may evidently have a favourable configuration for the evader. The time required by the evader to reach the chosen gate should generally be smaller than the time required by the pursuer to reach the same gate, in order for the evader to be caught, although this requirement is somewhat relaxed given the pursuer can reach an intercepting node with any arbitrary orientation. Furthermore, given the asynchronous scenario and forcing both the pursuer and the evader to complete every move before making a new one, there might be some cases when the pursuer is forced to head to a gate in one move, but the evader will change its target in the meantime. In such case, the pursuer will be stuck at the gate and the evader will be able to escape from the other exits. See Appendix B for example results.

At the end of the game, the paths are discretized with a fixed step and truncated at the point of collision, since the two robots were considered point-shaped and the actual collision between them is never checked at runtime.

References

[1] Johansen, Thomas. (2019). Autonomous Path Planning for Auto Docking of Ferries. 10.13140/RG.2.2.10105.21600.

[2] Shortest Path - Baeldung.com

[3] Yardimci, Ahmet Karpuz, Celal. (2019). Shortest path optimization of haul road design in underground mines using an evolutionary algorithm. Applied Soft Computing. 83. 10.1016/j.asoc.2019.105668.

A Results & Limitations

Depicted below there are several results obtained from the simulator. The dotted magenta line represents the path of the evader, while the dotted black line represents the path of the pursuer. Whenever the pursuer and the evader collide, the path is truncated and the simulation is stopped. The gates are depicted in green, while the obstacles in black. The red lines indicate the inflated obstacles for collision avoidance: see how the generated paths never intersect these lines if not to reach a gate.

Despite several successful trials in different maps, the proposed solution has some limitations. The disposition of the obstacles in the map plays an important role, as well as the random decisions made by the evader. In Fig[10] due to the proximity of the gate, with a free movement parameter of the robot equal to 5, the evader is able to reach the gate quicker due to its closer proximity. A possible solution in this case would be to grant the pursuer a greater degree of free movement by increasing the value of the parameter, in that case the probability of intercept are increased.

In Fig[11], a scenario occurred where the pursuer reached the last node before the gate faster than the evader and in this particular case as the evader had not yet finished its move from one node to another at that time, the pursuer planned its motion to go directly to the gate. In this case the randomness of the choices of the evader came to help as it chose to reach the same gate that the pursuer reached.

Overall, despite the pre-computation of several maps and the disposition of the obstacles such as in Fig.[7] where the number of visibility edges is higher compared to the other results, the overall timing performance of the algorithm can be said to be satisfactory, as shown in the single operations timing in the console screenshot in Fig[12].

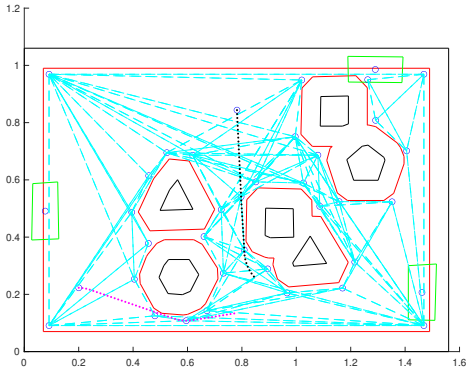

Figure 4: Example scenario: evader caught successfully.

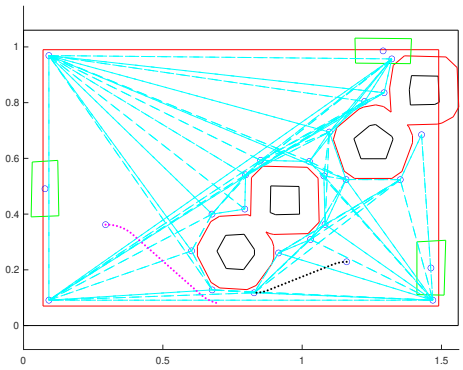

Figure 5: Example scenario: evader caught successfully.

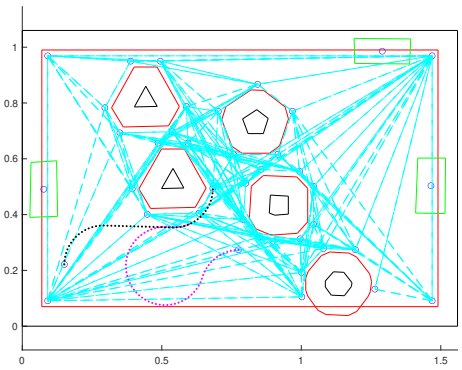

Figure 6: Example scenario: evader caught successfully.

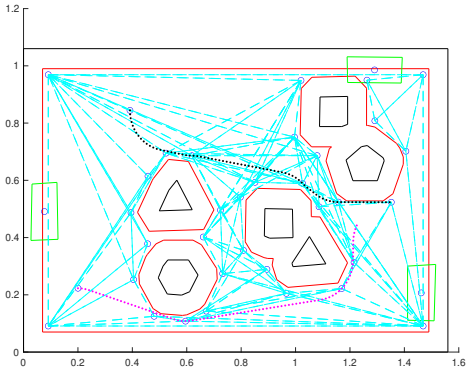

Figure 7: Example scenario: evader caught successfully.

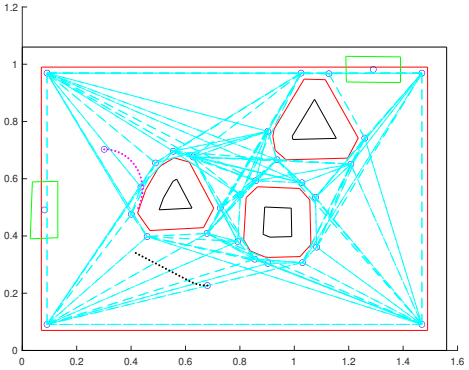

Figure 8: Example scenario: evader caught successfully.

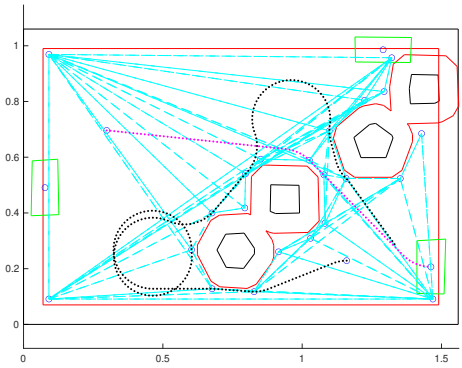

Figure 9: Example scenario: the pursuer is too far from the gate and the evader is able to escape.

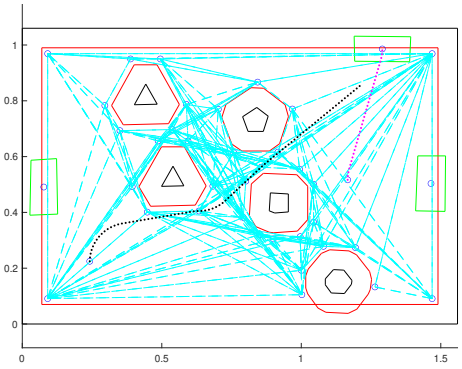

Figure 10: Example scenario: the evader has a favourable escape position.

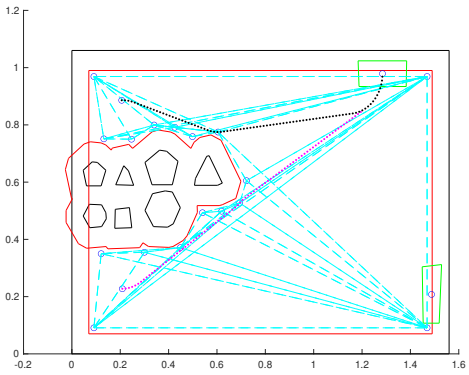

Figure 11: Example scenario: the pursuer reaches the gate before the evader.

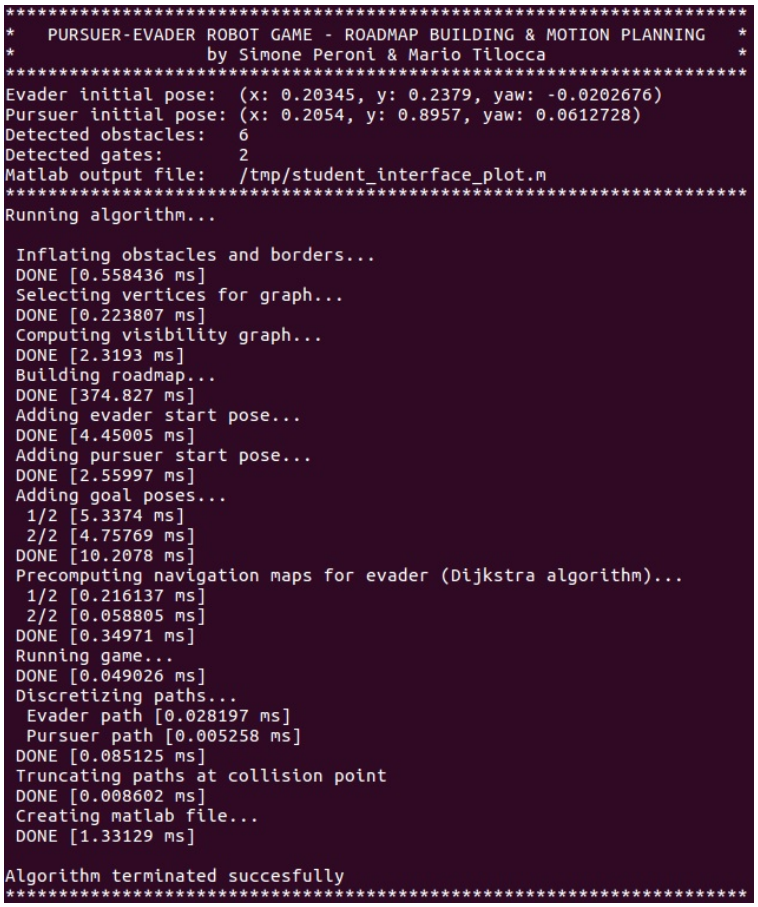

Figure 12: Console output for a map with 2 gates and 6 obstacles. Notice how the road-map building process is significantly longer than any other step.

B Influence of algorithm parameters

The algorithm can be tuned with some parameters that influence the complexity of the road-map and the chance of finding good solutions to the planning problem.

The adjustable parameters are the following:

- `collision_offset`: how much obstacles get inflated for collision avoidance.
- `visibility_offset`: how far from the obstacles the nodes for the visibility graph should be.
- `visibility_threshold`: the minimum distance between consecutive nodes of the visibility graph. Chunks of close nodes get averaged into a single one.
- `n_poses`: the number of orientations generated for each node.
- `kmax`: the maximum curvature allowed for the generated paths.
- `k`: when start and goal poses are added to the roadmap, they are connected to the k-closest nodes.
- `step`: discretization step.

In the following table, we summarize the default values and their influence on the solution.

Parameter	**Default setting**	**Influence of higher values**
`collision_offset`	$\frac{1}{2}$ `robot_size`	• Lower chance of hitting obstacles • Higher chance of occluding narrow passages
`visibility_offset`	1.3 `collision_offset`	• Higher chance of finding collision-free Dubins curves • Higher chance of occluding narrow passages
`visibility_threshold`	$\frac{1}{2}$ `robot_size`	• Lower road-map complexity • Higher chance of missing paths around obstacles
`n_poses`	8	• Higher chance of finding a feasible path • Final solution closer to the optimal multipoint Markov-Dubins problem • Greatly increased road-map complexity (quadratically)
`kmax`	1/`robot_size`	• Higher ability of the robot to follow the path at control time • Lower dexterity in narrower sections of the road-map
`k`	10	• Mildly increased road-map complexity • Higher chance of reaching far nodes in one move • Less nodes traversed in the final solution • High chance of evader escaping in a single move
`step`	$\frac{\pi}{32}$/`kmax`	• Higher ability of robot to adhere to maximum curvature arcs at control time • Mildly increased computing times

C Utility functions

C.1 Timer

A simple multi purpose timer has been developed to time the execution of single tasks in the main interface. The main functions are the following:

1. *tic*:
   which is used to create a new scope by starting the timer and optionally printing a message
2. *toc*:
   it stops the timer of the inner-most scope and it prints the entire duration in milliseconds together with an optional message.

C.2 MatlabPlot

This script has the purpose of creating a MatLab script *.m* file with all the instructions for plotting the results. This function has been used for the creation of the files displaying the results in Appendix B. The most important functions are the following:

1. *plotNodes*:
   function used to write the plot instructions for the nodes of the road-map
2. *plotEdges*:
   similarly, this function is used for writing the plot instructions of the edges of the road-map.
3. *plotPolygons*:
   this function is used for writing the plot instructions for a set of polygons such as the obstacles or inflated obstacles
4. *plotDiscretePath*:
   this function is used for writing the plot instructions for the discretized Dubins path of the pursuer and the evader.