Robot Planning and Automated Planning Joint Project
Pursuer-Escaper Scenario

Giovanni Lorenzini
223715
giovanni.lorenzini@studenti.unitn.it

Simone Luchetta
223716
simone.luchetta@studenti.unitn.it

Diego Planchenstainer
223728
d.planchenstainer@studenti.unitn.it

University of Trento, February 12, 2022

Abstract

A pursuer evader scenario with borders and obstacles in the map is a good case study to apply motion planning and automated planning to a real problem. Here, the pursuer has to catch the evader before it reaches one of the gates. To do so, the planning node of a ROS environment is coded. The information obtained by the processing of the map is exploited to encode the problem structure into a graph and then to encode the problem in PDDL. Metric FF is used to obtain the path which will then be refined by an iterative dynamic programming solution of the Dubins problem. The system is able to handle increasing evader behavioural complexities by planning a "fake" path of the evader and encode the information in the pursuer planner.

1. Introduction

The task consists into controlling a pursuer robot in order to catch an evader (fugitive) robot. The two autonomous agents are placed into a map, as shown in Fig. 1, in which are present other entities as borders, obstacles and gates. The entities are defined by their color: borders are black, obstacles are red and gates are green. Both robots have to respect some rules: borders and obstacles can not be touched, if this constraint is violated a penalty is added. The robots move at the same constant speed, this implies that neither of them has an advantage over the other.

The objective of the evader robot is to reach one of the gates without being caught by the pursuer. The pursuer, as stated before, has to catch the evader before it reaches one of the gates.
A ROS environment returns all the variables that are necessary to plan the paths that the robots will take. The environment is composed of different nodes but in this project, all the nodes, except the planning one, are taken for granted.

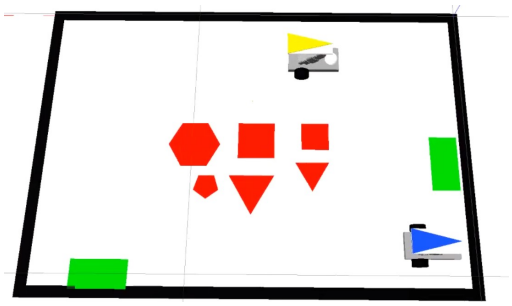

Figure 1. Example of a map structure

2. Understanding of the problem

The task boils down to control the robots inside the map; the pursuer has to catch the evader, the evader has to exit from the map by reaching one of the gates. The map is delimited by borders that can not be crossed and there are obstacles that can not be driven on, otherwise a penalty is added. A ROS environment is designated to control the robots and to elaborate the image obtained from the camera that point at the arena. Those elaborations return the following information: corner points of the borders, list of points that describe the obstacles, corner points of the gates and positions of the

robots (x, y, θ).
The focus of this work is to find the path that the robots will follow by using a planning approach.

Also, 3 different levels of behavioural complexity of the evader are taken into account:

- Level 1: there are two exits and the evader has to reach the nearest in minimum time.
- Level 2: there are two exits and the evader can decide non-deterministically at starting point to go in minimum time to one or to the other.
- Level 3: there are two exits and at each location of the roadmap the evader can decide to take one or the other accounting for the presence of the pursuer.

Some assumptions are made below:

1. The robots are initially deployed at random positions.
2. A ROS component will provide the exact position of the obstacles and of the robots in real-time (as precised above).
3. The evader, that is controlled by our code, moves through the same roadmap of the pursuer, but its information (gate towards which it is directed) are kept unknown to the pursuer.
4. When a robot takes a link to move to a location it has to complete its movement (i.e., it cannot stop or back off).
5. Robots move at constant speed. This allows to compute the path between two points through the resolution of the Dubins problem as discussed below in 3.3.1.
6. Synchronous behaviour: the system evolves in a sequence of states and each state is characterised by having the robot located in one of the nodes of the roadmap. In this way, every step of the plan is ended when all robots have reached a location and no robot is allowed to move to the next location before a step completes.

3. Design Choice

The implementation consists in three main parts: graph generation, planning, path computation.

3.1. Graph generation

Borders, gates, obstacles are obtained from the previous nodes and are processed here to extract meaningful information.

3.1.1 Obstacles and borders offsetting

First, borders and obstacles are inflated by a factor proportional to the dimension of the robot (the radius of the circle that circumscribes the robot), through the use of *clipper* library. This operation is accomplished to guarantee, at least in first place, that the robots will not drive over obstacles and borders. For every inflated obstacle, the convex hull is computed, to improve and simplify the map.

3.1.2 Cell decomposition

Later, the free space obtained by the previous operation is decomposed by an exact cell decomposition algorithm, trough the use of *CDT* library. This operation splits the space into triangles. We decided to stick to this approach over discrete sampling methods for several reasons.
First of all, the family of random exploration tree algorithms also perform some kind of search, hence lifting the demand of employing a planner to find paths solutions for our problem. Unlike combinatorial planning methods, which are complete, sampling based are not complete, even though they admit weaker notions of completeness (resolution and probabilistically complete), as the algorithm may run forever if a solution does not exists.
Note that since the size of the map in question is relatively small, one can afford to apply CDT without high computational costs. Then, other approaches would generate many more points that might blow up the computation needs for the planner. Though, we remark that it is still possible to further decompose the triangular cell map found via CDT if one wants to have a better refined granularity of the arena environment (but this is left for future works).
Overall, we chose exact triangular cell decomposition since: it is "strongly" complete, it has a low number of relevant points, it allows to apply a planner reasoning over it and it is easily applicable in this scenario.

3.1.3 Graphmap generation

From the decomposed map are then extracted the centers of the triangles and the midpoint between the edge that connect two triangles. With the information recovered by the triangles is possible to construct the graph structure that will encode the properties of the problem. The graph structure is build upon the *Boost* graph library. The choice fell on *Boost* because it: is open-source and peer-reviewed, complements STL rather than replacing it, is well documented and provides an elegant and efficient platform independent to needed services.

The two main elements that compose every graph are vertexes and edges. In vertexes are stored the waypoints that are extracted from the decomposed map, every waypoint has two attributes: coordinates and type. The type attribute has been added to differentiate simple waypoints from gates and robots locations.
The edges contain the euclidean distance between the two vertexes that they connect.
As an optimization step, it has been chosen to connect vertexes at distance 2. For distance 1 are intended adjacent vertexes connected by an edge. For distance 2 are intended pair of vertexes that are connected to a common vertex. This is done to give the possibility to use a smother path.
The overall result of this step is reported in Fig. 2.

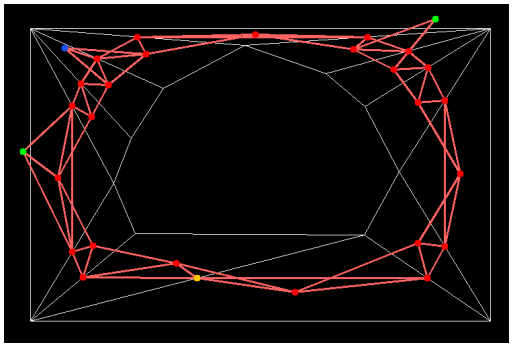

Figure 2. Example of a graph structure: white lines are the result of the exact cell decomposition, red dots are vertexes in the graph and pink lines are edges that connect the vertexes. Green dots are the center of the gates and blue and yellow dots are robots.

3.2. Planning

To establish which path the robot will take, an approach based on a planner is chosen. The problem is formulated in Planning Domain Definition Language and fed to a planner that returns the decided path which the robot will execute.

3.2.1 Evader planning

Level 1 evader chooses the minimum cost path that reaches the closest gate.

Level 2 evader finds the minimum cost path that reaches one random gate in the arena. Once the gate is established, the evader commits to that decision.

For evader level 3 complexity the implementation is the following: the evader-gates distances and pursuer-gates distances are computed with the "distance planner". Then the evader chooses the gate which is nearest to the evader, but farther from the pursuer and compute a plan to reach it, as in Eq. 1.

$$Gate = \min_{G_1,\dots,G_n} \left[\frac{dist(E, G_j) - dist(P, G_j)}{dist(E, G_j) + dist(P, G_j)} \right] \quad (1)$$

Where $G_1, \dots, G_n$ are all possible gates appearing in the arena, E and P represent the evader and pursuer robot respectively.

3.2.2 Pursuer planning

Since the evader can behave in 3 different ways, the pursuer need to approach every scenario accordingly.

Regarding the first level of complexity, the pursuer plan is found in two steps. Firstly, a PDDL problem file is fed to the planner, so to estimate the evader's shortest path to the closest gate. Secondly, this information is exploited by the pursuer, that knows the estimated path to the gate of the evader. The estimate of the evader's route is simply a list of tuples that contain the waypoint and the length of the path that the evader took to reach it. The pursuer planner searches for a solution that minimizes the travelled distance and reaches a point in which the pursuer robot travelled distance is lower than the evader's. If it is not possible to catch the evader, the pursuer moves towards the gate in which it believes the evader is heading to, to be ready in case of an evader's mistake.

For level 2 and 3 of complexity, the approach is to evaluate the distances of the evader with respect to the gates trough the use of the so called "distance planner" at every step. This is done because the objective of the evader is unknown and needs to be estimated. The gate towards which the evader is going for the pursuer belief is decided by doing the subtraction between the distance at the previous step and the distance at the current step for each gate. The biggest result is the gate in which the pursuer believe the evader is going. Furthermore, the algorithm accounts (in a decaying way) for the history of the distances of the evader robot with respect to the gates. These measures are stored after each iteration and are kept within a file.
Beside from the part used to estimate the evader path, the planner acts as in level 1 explained before.

3.2.3 Domain file

The planner is called trough a shell command by the program and requires a domain file and a problem file in or-

der to work. The domain file is written and it remains unchanged, instead the problem files are generated by the C++ application at every execution.

The structure of the domain file is composed as follows:

- Requirements inclusion: the core of the problem encoding is all about finding the minimum distance cost of a path, hence *action-costs* requirement is of prioritary importance. Numeric fluents are declared in the appropriate *functions* block, and their matching value is modified according to the effect section included in the *action* blocks.
  To let higher-level concepts to be automatically derived from other propositions, *derived-predicates* is also included. This permits to a predicate to become automatically true whenever a particular quantified precondition is full-filled.
- Types and constants: there are a total of 6 types specified in the domain. The *robot* with its two subtypes: *evader* and *pursuer*; and the *location* with its subtypes: *gate* and *waypoint*.
  Two constant objects are instantiated: *r1* as pursuer and *r2* as evader robot.
- Actions: the main action a robot can perform is *move*. Depending on the label of the robot, a matching identificative predicate is turned either on or off. The robot is allowed to move from a location to another one only if those are *near* to each other. As an effect of this, the fluent *total-cost* is increased by an amount equal to the *distance* between the two locations.
- Predicates:
  - *Near*: specifies whether there is a link that connects one location to another.
  - *At*: describes where a robot is with respect to a location.
  - *Caught*: derived predicate that is automatically turned to true if there exists a location in which the pursuer robot is at, and it reaches that location before the evader robot does.
  - *Evaded*: derived predicate that is assigned to true if the evader robot is at the location corresponding to a gate.
  - *Pursuing*: enables the pursuer to move.
  - *Evading*: enables the evader to move.

3.2.4 Problem files

The key elements that compose the problem file are presented hereinafter:

- Objects: definition of the object comprehend naming all the locations that can be of type waypoint or gate, from information that is extracted from the graph.
- Initialization: first of all, the metric *total-cost* to be minimized is initialized to zero. Then, starting positions for both robots are encoded.
  The whole structure of the graph is made explicit by stating for each vertex the triplet composed of *near* predicate between neighbouring locations and the pairwise mutual distance between the two, that is the information contained in the edge.
  Lastly, an additional predicate sets the type of the problem, either evading or pursuing.
  If it is a pursuer problem, then the *evader-cost* predicate is set for every position in the estimated evader path to the distance that the evader took to reach that position. For locations excluded from the estimate of the evader's path and for the current location of the evader, *evader-cost* is set to -1 because so they cannot be a location part of the solution of the pursuer. The believed destination of the evader is set to a very high value to be chosen by the pursuer if other options are not available.
- Goal: a solution is found whenever the matching derived predicate (*caught* or *evaded*) is set to true. For the evader, the robot is also constrained to be at one arbitrary gate that is chosen depending on the behavioral complexity.
- Metric: The planner minimizes the plan's length in terms of total incremented distance over the path's points.

3.2.5 Metric FF

Since it is crucial that the plan is optimal in order to maximize the possibility to succeed, the planner used is Metric FF 2.1 [3] as it can minimize and handle numerical values, such as distances in this case; it is important to stress out the fact that Metric FF can deal with lesser-greater operators, whereas most of other planners cannot.
Metric FF can be called with the following console command:

```
./ff -o domain.pddl -f *.problem.pddl -s 3 -w 1
```

The formulation above allows to use the option `-s` to specify the search strategy: 3 is going to be weighted A* search with weight specified by the option `-w`.
Metric FF takes in two files, namely `-o`: operator and `-f`: fact, that are the equivalent for "domain" and "problem". The symbol * in the upper command denotes the problem file that is respective of the type of problem to be solved, hence either *pursuer*, *evader*, *distance*, *evader_estimate*.

3.3. Path computation

The path extracted by the planner has to be refined to be used by the robots. Since the robots move at constant speed, one possible approach is to solve the Dubins problem between two points extracted by the planner. Since the boundary condition are not fixed for all the point except the first, it is necessary to compute them. The approach here implemented leverage on Iterative Dynamic Programming to extract the parameters needed as proposed by Frego *et al*. [2]. The Dubins path so obtained is forwarded to the next node that will control the robot.

3.3.1 Dubins problem

As assumed in Section 2 the robots move at constant speed and can travel only forward; furthermore, they have a fixed maximum steering angle k, which results in a minimum steering radius ρ. These assumptions allow to compute the path through the resolution of the Dubins problem. This problem aims at minimizing the length of the curve that connects two given points p_i, p_f, expressed as (x, y, θ), where θ stands for the orientation of the robot. It was shown that between any two configurations, the shortest path for the Dubins problem can always be expressed as a combination of no more than three motion primitives, expressed as L for left turn, R for right turn and S for straight line. Each motion primitive applies a constant action over an interval of time. Dubins showed that only 6 combination of 3 motion primitives can be optimal: $LRL, RLR, LSL, LSR, RSL, RSR$. Each of these combination is called Dubins curve.

3.3.2 Iterative Dynamic Programming approach to Dubins problem

The core of the idea is to solve multipoints Dubins manoeuvres iteratively to find the minimum length path that interpolates all the waypoints found by the planner. The problem is formulated in terms of minimization over the set of angles for which the Dubins maneuvers yield a corresponding total distance L to be minimized as in Eq. 2.

$$L(j, \vartheta_j) = \min_{\vartheta_{j+1}} D_j(\vartheta_j, \vartheta_{j+1}) + L(j+1, \vartheta_{j+1}) \quad \text{for } j = n-1, \dots, 0 \tag{2}$$

Each point in the path has a matching angle that belongs to the set $\Theta = (\vartheta_0, \dots, \vartheta_n)$, where n is the length of the path. Each angle is discretized with a resolution k, leading to a granularity of $h = \frac{2\pi}{k}$.

The idea is to let k be deliberately small to reduce the computational complexity of the algorithm and find a first coarse approximation over a broad interval of radians angles, then refine the multipoints Dubins solution over m iterations by sampling k angles around ϑ_j in the range $\left[\vartheta_j - \frac{3}{2}h, \vartheta_j + \frac{3}{2}h\right]$. This leads to an improved granularity that spans around the best angle ϑ_j of the previous iteration.

For each iteration step i and point j, given the angle index $\vartheta^d_{j,i}$, its corresponding radians value ϑ_j is recovered as in Eq. 3.

$$\vartheta_j = \sum_{i=0}^{m-1} \left[\vartheta^d_{j,i} \frac{2\pi}{k} \left(\frac{3}{k}\right)^i - \frac{3}{2} \frac{2\pi}{k} \left(\frac{3}{k}\right)^{i-1} \frac{k-1}{k} \right] \tag{3}$$

Instead, given an angle in radians ϑ_j, it is possible to recover the matching indexes $\vartheta^d_{j,i}$ forming up the angle by following Eq. 4.

$$\vartheta^d_{j,i} = \frac{\theta_j}{h} + \sum_{l=0}^{i-1} \left[-\vartheta^d_{j,l} \left(\frac{3}{k}\right)^l + \frac{3}{2} \left(\frac{3}{k}\right)^l \frac{k-1}{k} \right] \tag{4}$$

3.3.3 Collision check

For every Dubins solution encountered, a collision check algorithm is run. If a collision happens, a penalty for that particular angle is added to disincentivate the algorithm to take that solution. Anyway, it is clear that this approach only searches for the optimal angles, not for the optimal Dubins solution that does not collide. On the other side this approach is more computationally efficient and it has been chosen in place of a penalization inside the computation of the Dubins curve.

3.3.4 Moving the robot

Once the optimal angle for every waypoint in the path is found, a combination of Dubins manoeuvres is computed for each robot. Then, this information can be used to retrieve the vector of poses that is sent to the ROS node responsible for movements control.

4. Results

All tasks introduced in the previous sections are full-filled. Movements for each robots are planned according to information contained in a graph representing a roadmap that is extracted by applying the CDT algorithm. The input for controlling the motion is sent to ROS once Dubins manoeuvres are computed and sampled.

It might be worth showing some testing scenarios data we collected by deploying the system on modified versions of the map, with various degrees of complexity. Images and descriptions can be found in the Appendix. Furthermore, here is the link to the Google Drive folders containing our recordings of the tests we conducted.

4.1. Known issues

Note that we are aware of some small flaws in the system:

- We noticed that the controls applied to the robots are not always executed precisely. For example, once a Dubins manoeuvre has been accomplished, the angle and the position at which the robot arrives at the destination point might differ from what had been computed in previous Dubins steps. This disrupts the motion plan for the steps that come afterwards.
- Abrupt movements: sometimes the robot spins around itself at huge speed. We do not know why this happens.
- Camera fail: the detection of the position of the robot on the ROS node responsible for image acquisition returns *false*. This can happen in two circumstances: the first one when the robots collide (which is to be expected, since one of them goes underneath the other), and the second one which is yet unclear, but we think that is due to the shallow contrast that the yellow color of the second robot provide. So we changed it to purple and it seems to work better.
  Once this happens, it leads to the impossibility to control the robots, that will keep their moving direction straight, thus exiting the arena.
- The roadmap that is obtained via CDT might be not precise enough: one fix for this could be to further decompose each triangle to generate new smaller ones, for which it will be possible to have new barycenter and middle points that help increasing the granularity of the map. This improvement could help the planner to find a shorter plan.
- Unfair step size: suppose that the pursuer can connect two points in the roadmap which are more far apart with respect to those for the evader. This is unfair because the pursuer can clearly cover more distance in a single step. This problem can only be resolved by refining the roadmap, or by placing an unitary distance constraint for driving the robots movements. We tried to add the second, but we found out that it worsened the results, as the robots might end up on a point in the map where messy connections in the graphmap will then be made. To fix this, we even tried to place a link between each robot and the way-points in line of sight. However, so-found solutions were inaccurate in our opinion and computationally demanding, so we discarded them. As an aside, this approach needs to find whether a point intersects with an obstacle or not, slowing up the algorithm.

Table 1. Number of way-points to connect and time elapsed. Here, an angle resolution of $k = 20$ and iterations $m = 3$ are used.

Number of Points	Time [ms]
6	760
8	850
10	1150
12	1230
38	3650

4.2. Computational time constraint

Other minor constraints are given by the computational time. As the path of the robots is recomputed at every step and a game consist of several steps, the computation time should be kept low. This lead to a suboptimal result from the Dubins algorithm (not always more iterations make up for the low number of subdivisions of the angle). Another problem is to not being able to fully modify the path to avoid the obstacles, however the Multipoints Dubins algorithm try different angles for the points to avoid touching an obstacle. Examples have shown that the computational time varies depending on the number of waypoints that need to be connected by multiple manouvres; for instance, the time taken for up to 4 or 5 way-points is only of $500ms$ thanks to the iterative Dubins algorithm (angle resolution $k = 20$, iterations $m = 3$). Results are available in Table 1.

Tests were conducted in order to detect obstacle intersections when performing Dubins manoeuvres, penalizing the solutions that collided with elements in the arena. In this scenario, the length of the so found solution would be increased by a factor proportional to the number of intersections. This approach yields to a suboptimal Dubins route, yet clear of intersections with borders or obstacles. Though, since this check is done inside iterative Dubins, it affected the computation times, slowing up by a lot. In the end this approach indeed modified some trajectories, sometimes helping them being more precisely constrained in the arena

borders, but at the cost of introducing major computational overhead. This is also the reason why this kind of check on obstacles intersections is not deployed.
On the other hand the exact cell decomposition done by triangulation provides a roadmap that is very similar to a maximum clearance roadmap and thus the probability of touching an obstacle is reduced given the fact that the robot is performing a Dubins manoeuvre that has to necessarily pass through the so-found waypoints.

4.3. Fulfilled objectives

Otherwise, the code we provide is perfectly functioning and achieve all the requirements:

- the robots move at constant speed without (almost) touching obstacles and walls;
- three levels of behavioural complexity have been implemented;
- a PDDL planner is used for planning the movements of the robots;
- the evader (tries to) escape through a gate;
- the pursuer (tries to) catch the evader.

4.4. Code organization

The code is organized as below:

```
workspace/
└── project/
    └── src/
        ├── boost/
        ├── CDT/
        ├── clipper/
        ├── dubins/
        │   ├── darc.hpp
        │   ├── dcurve.hpp
        │   ├── dpoint.hpp
        │   ├── dubins_utils.hpp
        │   └── dubins.hpp
        ├── pddl/
        │   ├── domain.pddl
        │   ├── distance.problem.pddl
        │   ├── evader_estimated.problem.pddl
        │   ├── evader.problem.pddl
        │   ├── pursuer.problem.pddl
        │   └── ff
        ├── planner/
        │   ├── planner_distance.hpp
        │   ├── planner_evader_estimate.hpp
        │   ├── planner_evader.hpp
        │   ├── planner_pursuer.hpp
        │   └── planner.hpp
        ├── cell_decomposition.hpp
        ├── convex_hull.hpp
        ├── graph_map.hpp
        ├── line_offsetter.hpp
        ├── main.cpp
        ├── router.hpp
        └── student_interface.cpp
└── simulator/
```

In *simulator* is contained the ROS environment at the basis of this project forked from https://github.com/AlexRookie/AppliedRoboticsEnvironment [1]. Instead, in *project* is contained all the code that is necessary to run the planning node. We choose to implement all the code in an header-only library fashion.
The folders *boost*, *CDT* and *clipper* are library folders that are used in the code. Instead, the other *.hpp* files are part of our code.

A brief explanation of the content of the files:

- *dubins* folder contains the code necessary to compute a single Dubins maneouvre between two points.
- *pddl* folder contains *domain.pddl* that is the domain for all the PDDL problems, the automatically generated problem files (**.problem.pddl*) and the executable of Metric FF (*ff*).
- *planner* folder contains the C++ files used to generate and read the PDDL problems. In particular, *planner.hpp* contains the parent class of the others and the shared code for dealing with the planner. The *PlannerDistance* is used to compute the distance between two points in the map. The *PlannerEvaderEstimate* tries to estimate the path that the evader is taking. *PlannerEvader* and *PlannerPursuer* are the planners of the two respective robots.
- *cell_decomposition.hpp* contains the code for doing the cell decomposition using the *CDT* library.
- *convex_hull.hpp* contains the code that make the convex hull of the obstacles.
- *graph_map.hpp* contains the code for storing and retrieving in a graph the roadmap information like waypoints, gates and the robots.
- *line_offsetter.hpp* deals with the offsetting of the obstacles and the borders.
- *main.cpp* is used for testing purposes only.
- *router.hpp* contains the code for doing multipoints Dubins using the iterative technique presented in [2].
- *student_interface.cpp* instantiate the other classes and call the appropriate methods for planning the movements of the robots.

5. Conclusions

In this report a pursuer-evader scenario is faced to implement in practice motion and automated planning concepts. First, obstacles are inflated, then the convex hull of the obstacles is obtained and finally the C_{free} space is decomposed in triangles. The graph is extracted from the salient points of the decomposed map and these information are encoded in the PDDL problem. Metric FF is called to plan the path and then the actual trajectory is computed by resolving multipoints Dubins problem through iterative dynamic programming.

The implemented solution is good in terms of computational time and in the most of the cases behave well. It can navigate robots inside the map without them touching the walls most of the times, leverage on Metric FF planner and usually the pursuer is able to catch the evader, at least in behavioural complexity 1 and 2. Anyway, some flaws are present such as the graphmap precision that could be enhanced, the so called "unfair" path condition that could be handled and some errors that we think are fault of the ROS environment (camera fails, abrupt movements and not precise control).

One possible future work could go in the direction of analyzing a better way to decompose the map, as the results that have been found are strictly correlated with the way the collision-free space is computed. Since in evader complexity level 2 and 3 the objective of the evader is not known, one possible future implementation could try to embed a non deterministic planner instead of using hard-coded workarounds.

References

[1] AlexRookie. AppliedRoboticsEnvironment. https : / / github . com / AlexRookie / AppliedRoboticsEnvironment, 2021. [Online; accessed 08-January-2022]. 7

[2] Marco Frego, Paolo Bevilacqua, Enrico Saccon, Luigi Palopoli, and Daniele Fontanelli. An iterative dynamic programming approach to the multipoint markov-dubins problem. *IEEE Robotics and Automation Letters*, 5(2):2483–2490, 2020. 5, 7

[3] Jörg Hoffmann. The metric-ff planning system: Translating "ignoring delete lists" to numeric state variables. *CoRR*, abs/1106.5271, 2011. 4

A. Appendix

Here below are presented some example to show the behaviour of the system. All the complexities will be tried in two different maps, the first has been given by the professors and presents 6 obstacles in the middle of the map. The other was constructed by us and has a more challenging disposition. All images presented below will show the position of the robots in the simulator and their planned path in the black map, where blue path stays for pursuer's and yellow's for evader's.
In all future cases the first image will show the initial configuration of the robots and their planned path, so the caption will be omitted.
As listed above we recall that the evader robot is coloured in pink, but its path is highlighted in yellow. Collision points with obstacles are coloured in light blue.

A.1. Complexity 1 - Given map

At complexity 1 the path is executed in its integrity without any stops in the middle. In Fig. 3 it can be seen that the pursuer is able to catch the evader since it starts in an advanced position, otherwise it would not.

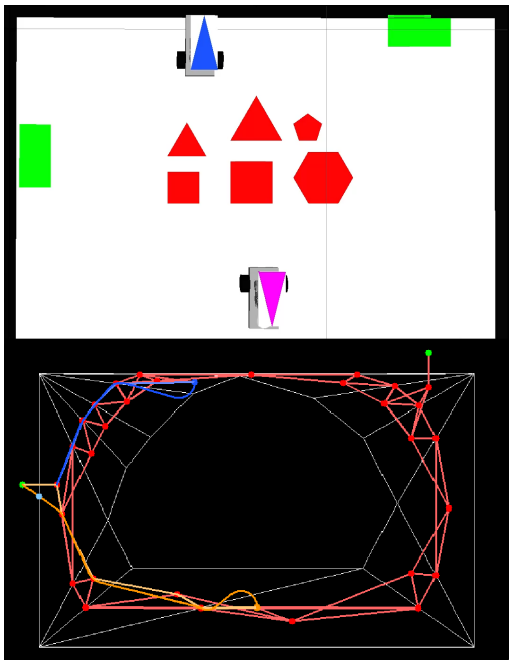

Figure 3

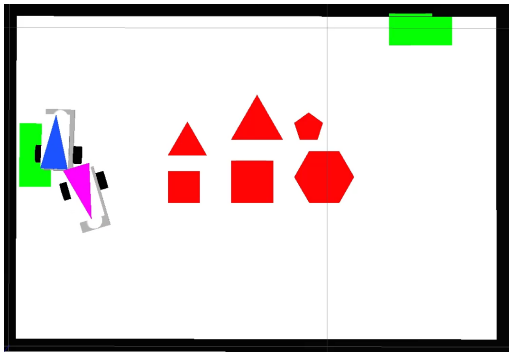

Figure 4. Final configuration, pursuer wins.

A.2. Complexity 2 - Given map

The behavior of the pursuer is as follows: initially, since the pursuer cannot know to which of the gates the evader will escape to, it estimates that the best possible escaping route for the evader will be the one towards the nearest gate. Though, the evader chose a random gate. So only after one step, the pursuer can understand where the evader is heading by looking at the differences between the current and previous positions, hence tries to intercept the opponent before it reaches the gate. In Fig. 5 and Fig. 6 are shown two different runs of the code in which it can be seen that the chosen gate changes. In Fig. 7 it can be seen that the pursuer rethinks its path to reach the evader. Here, because of the above called "unfair" path condition, the evader manages to escape, as can be seen in Fig. 8.

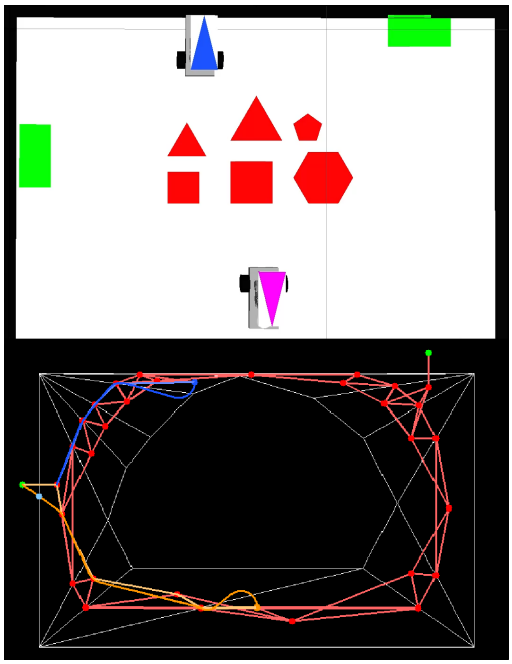

Figure 5. The evader decides to go non-deterministically to the left gate.

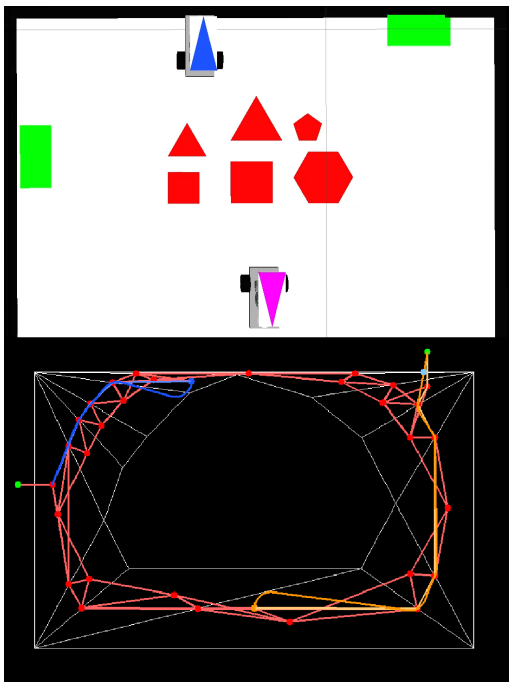

Figure 6. The evader decides to go non-deterministically to the upper-right gate.

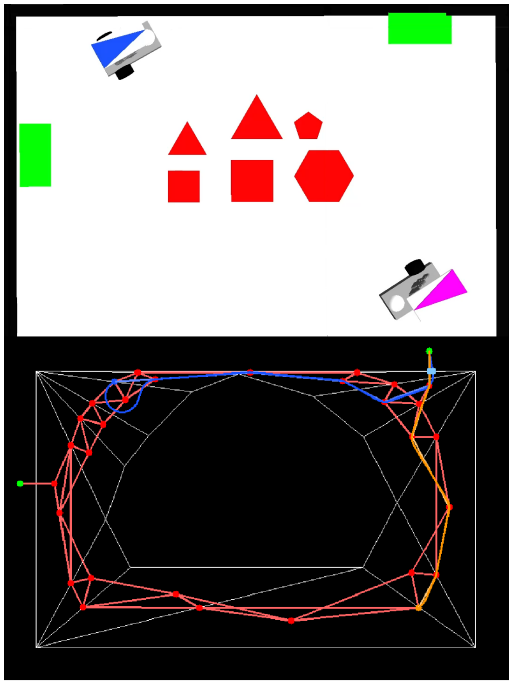

Figure 7. The evader has decided to go non-deterministically to the upper-right gate. The pursuer rethink its path to intercept the evader.

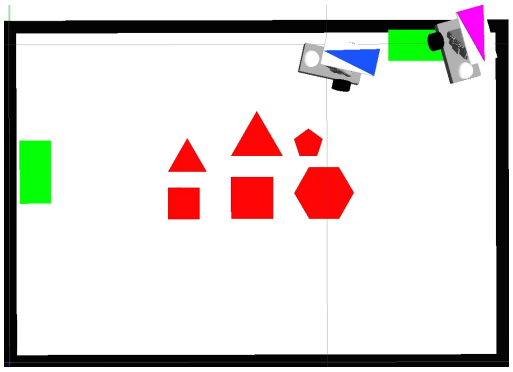

Figure 8. Final configuration, evader wins.

A.3. Complexity 3 - Given map

Planners are called for both robots. The pursuer behaves as in level 2 of complexity. Though, the evader chooses to go to the gate that is the biggest in ratio with respect to the difference of distances of both robots to the respective gates, as in Eq. 1.

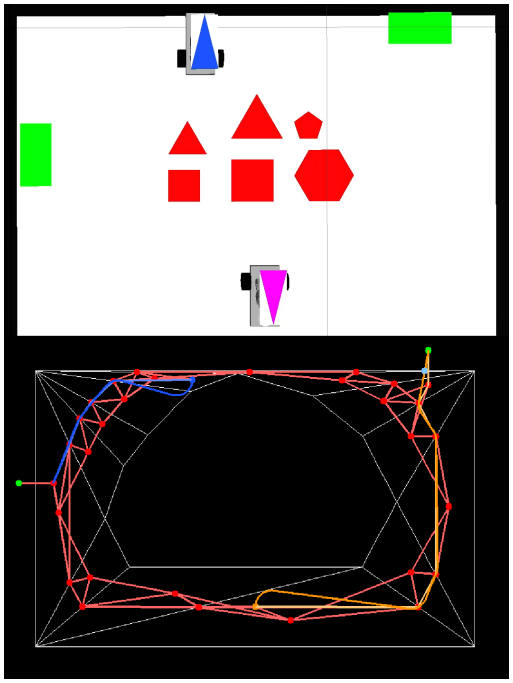

Figure 9

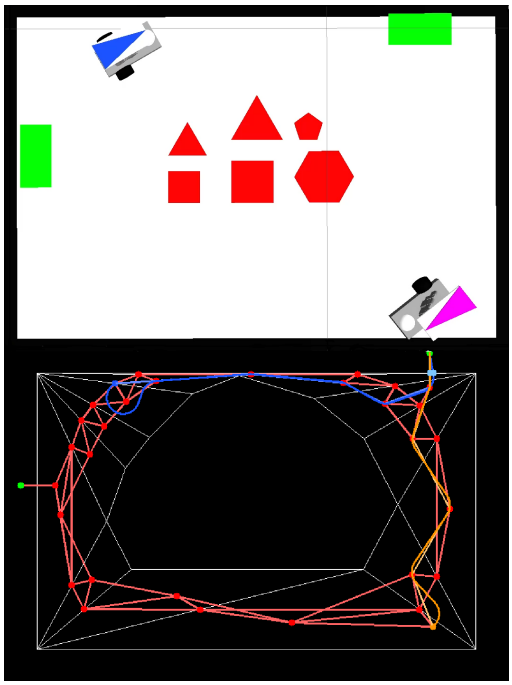

Figure 10. Robots configuration after one step (upper). The planner returns a path for both robots improving the behavior of the pursuer (below).

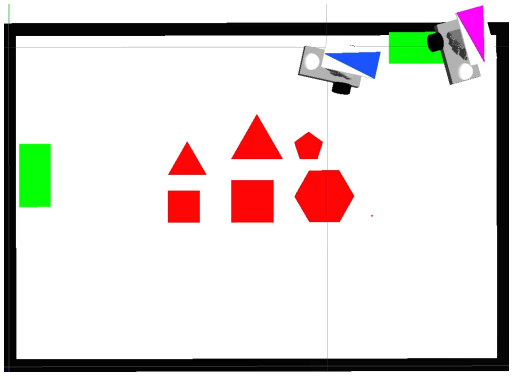

Figure 11. Final configuration, evader wins.

A.4. Complexity 1 - Map with passage on the bottom

At complexity 1 the path is executed in its integrity without any stops in the middle. In Fig. 12 it can be seen that the pursuer is able to catch the evader since it starts in a advanced position, otherwise it would not.

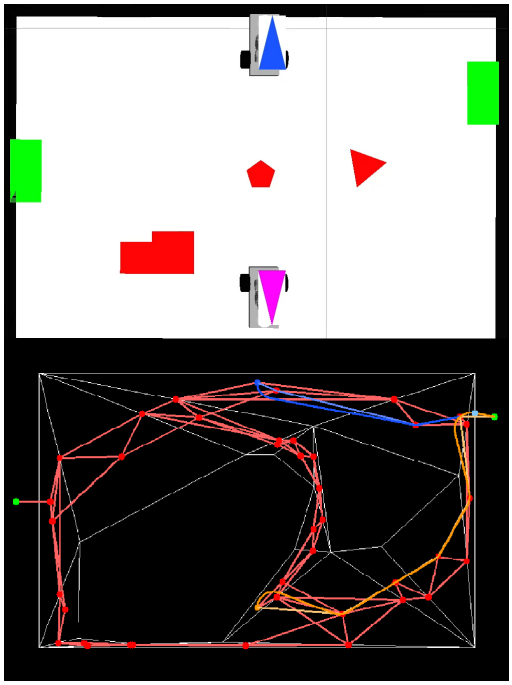

Figure 12

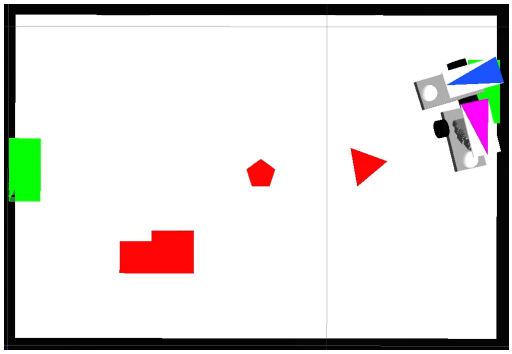

Figure 13. Final configuration, pursuer wins.

A.5. Complexity 2 - Map with passage on the bottom

The robots here behave as in A.2.

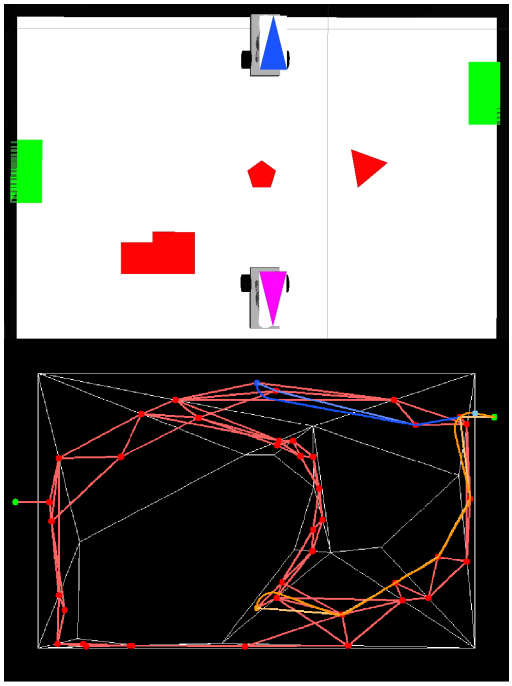

Figure 14. The evader decides to go non-deterministically to the right gate.

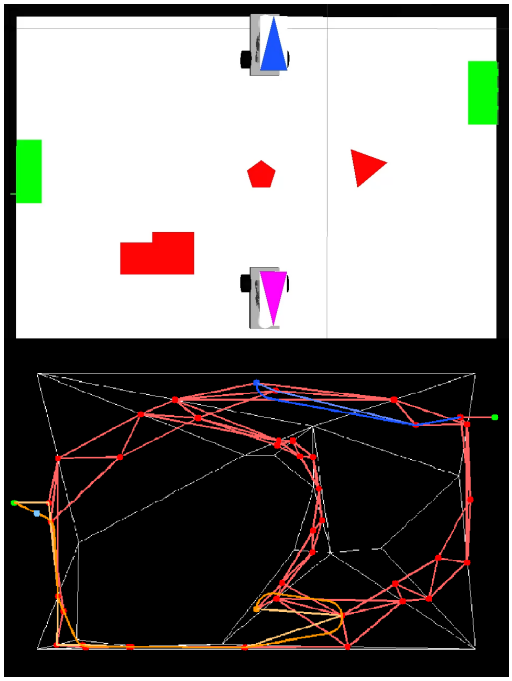

Figure 15. The evader decides to go non-deterministically to the left gate.

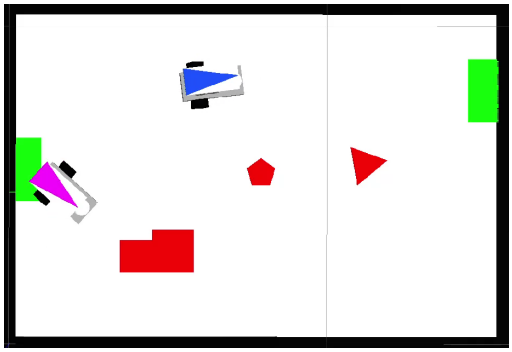

Figure 16. Final configuration of Fig. 15, evader wins.

A.6. Complexity 3 - Map with passage on the bottom

Robot plans for level three complexity are presented in the following figures. They behave alike in A.3.

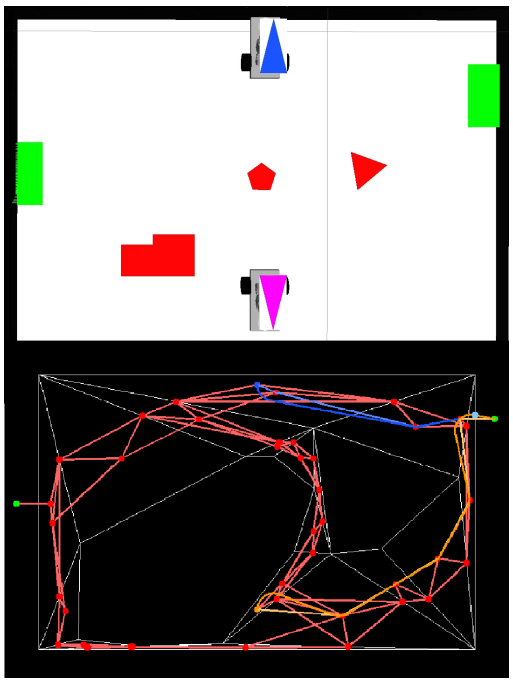

Figure 17. Starting configuration and matching plan.

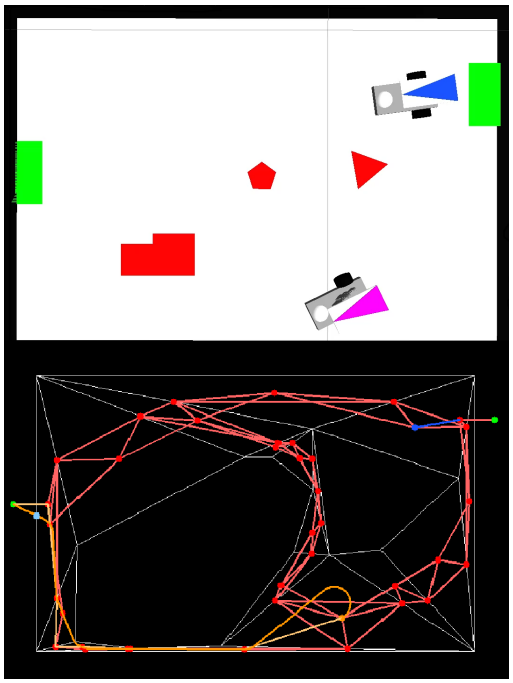

Figure 18. The evader rethinks its plan.

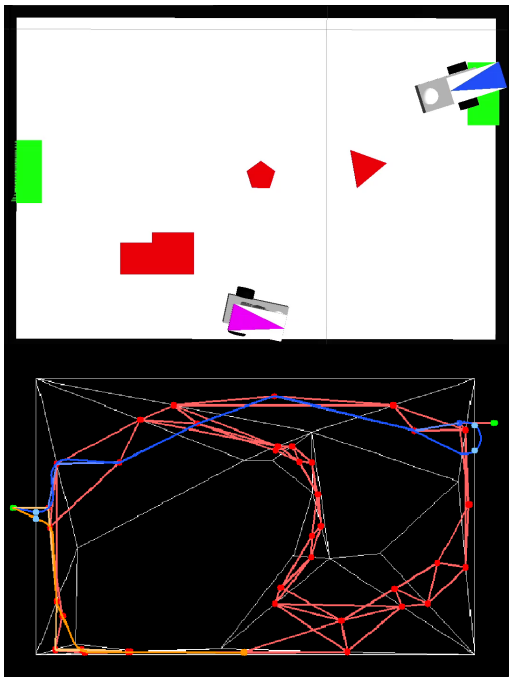

Figure 19. Robot configuration after yet another step and matching plan. The pursuer rethinks its plan. Intersection of the pursuer path with the obstacles are shown in light blue.

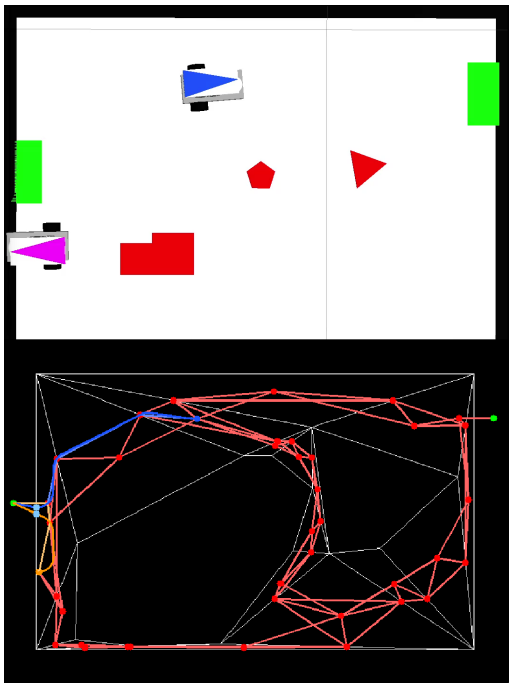

Figure 20. Robot configuration after more several steps and matching plan. The evader performs its last plan computation, along with the Dubins manoeuvre. Pursuer will not be able to reach the evader.

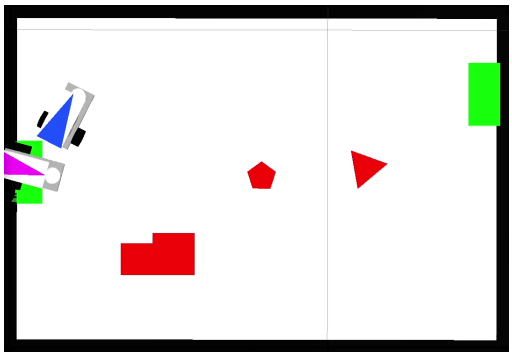

Figure 21. End of the run: the evader is at the gate. The pursuer has lost.