Pursuer-Fugitive Robots Simulation

Davide Bertelli
223718
University of Trento
davide.bertelli-1@studenti.unitn.it

Davide Dalla Stella
223727
University of Trento
davide.dallastella@studenti.unitn.it

Abstract

A pursuer and a fugitive robots are placed inside an arena characterized by the presence of obstacles and exit gates. The pursuer's goal is to catch the fugitive before it escapes from any possible exit. This scenario is very suitable for studying the applications of motion planning and automated planning in real problems. All the information about the environment are processed to code a PDDL problem that the Metric-FF planner uses to determine the path that a robot must carry on to achieve its goal. Subsequently this path is then refined into a dubins path to be followed from point to point.

1. Introduction

The goal of this work is to allow a pursuer robot to intercept a fugitive one before it escapes from a possible exit in the arena. These robots are placed in a map like the one shown in Fig. 1.

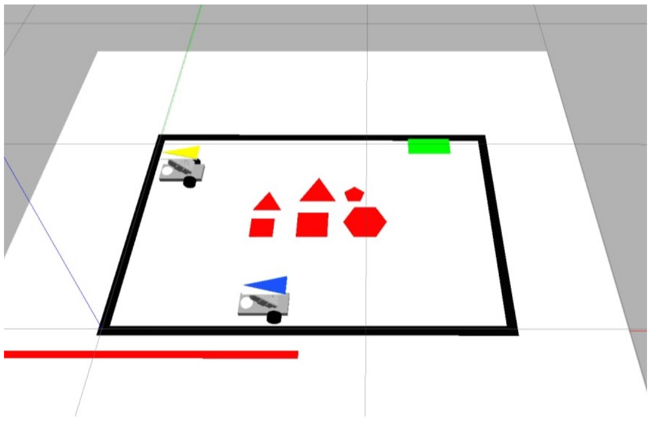

Figure 1. Example of the arena

Note that the various objects are color-coded:

- Red: obstacles inside the arena.
- Green: exit gates.
- Black: limits of the arena.
- Blue triangle: fugitive robot.
- Yellow triangle: pursuer robot.

A fundamental rule to follow in addition to completing the task is to make sure that no robot collides with the limits of the arena and with the obstacles.

More specifically the goals of this work are:

- development and implementation of intelligent planning algorithms to decide the optimal game strategy.
- development and implementation of motion planning algorithms to move the robot from an initial to a final position in minimum time and/or space avoiding obstacles.
- testing of the solution through simulations.

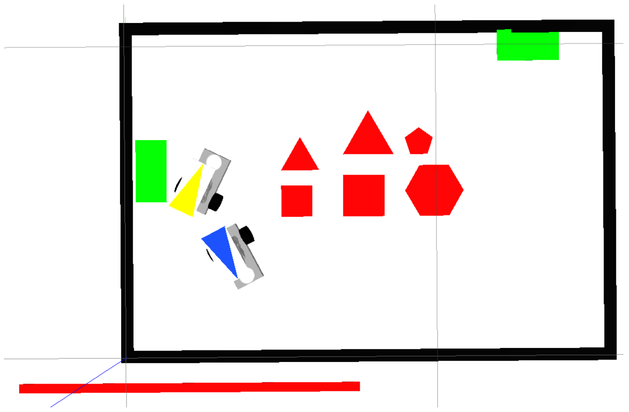

Figure 2. Example of winning task

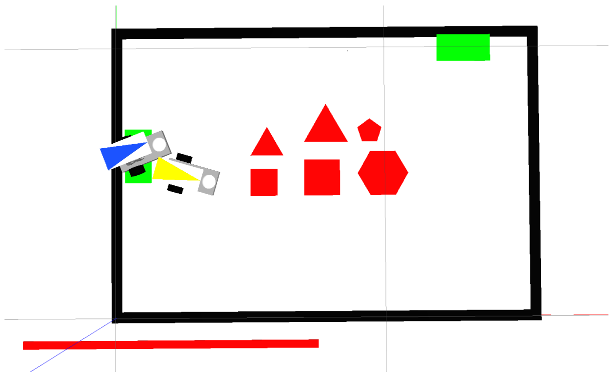

Figure 3. Example of failing task

2. Problem Statement

All the information about the objects in the arena and the robots position (x,y,th) are given by the ROS simulator in use. The purpose of the project is to process all the information provided to be able to obtain the paths that the robots must follow. This work can be divided in three sub-tasks:

- Roadmap generation.
- Paths planning.
- Dubins paths computation.

3. Roadmap generation

The roadmap generation task has been carried on through several progressive steps which allowed to describe the environment of the arena in a suitable way for the agents. In particular the arena and the problem solution have been addressed from the point of view of a 2D geometrical problem. Following this setup the agents have been represented as material points, while the free space areas, the gates and the obstacles have been considered as polygons.

3.1. Obstacles and Gates Management

To enforce the constraint of no collisions the limits of the arena have been shrunk and the obstacles have been enlarged by a quantity equal to the supposed dimensions of the agents. The gates have not been affected by such considerations and following a pessimistic lead they have been reshaped whenever an obstacle overlaps with them. After being enlarged, the obstacles touching one another had been merged into a unique entity and if a concave result is produced it has been substituted by its convex hull to simplify the computations of the dubins path and account for a more straight forward solution. A possible outcome of the obstacles transformations can be seen in figure Fig. 4.

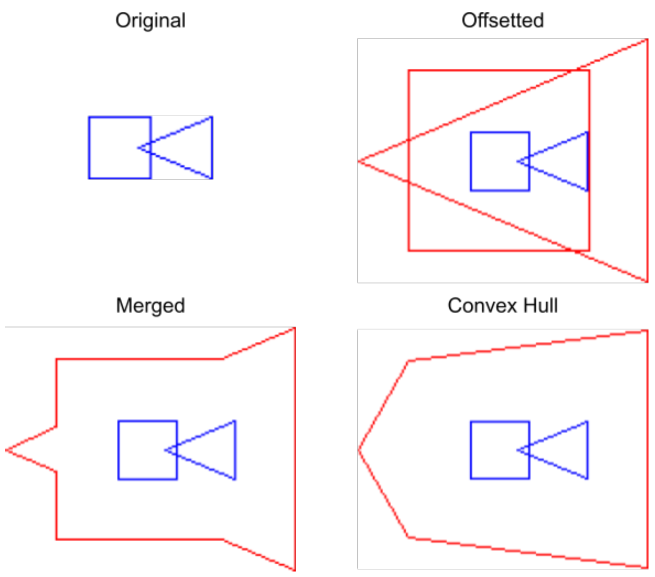

Figure 4. Example of obstacle management.

3.2. Free Space Generation

For the generation of the free space, two methods of arena decomposition have been implemented, tested and compared. The used methods are:

- Vertical Exact Cell Decomposition - Trapezoidal Decomposition.
- Approximate Cell Decomposition.

3.2.1 Exact Cell Decomposition

This algorithm uses the vertical sweep lines generated by the vertices of the obstacles to create free space cells. Each line is defined from the top of the arena to the bottom and are used to divide the arena space into n cells. Once the cells have been created, the area of the obstacles is subtracted from each of them to obtain only cells containing free space and perfectly adjacent to the obstacles. At the end any pair of cells that share a side not generated by a vertex are merged. The main advantage of this approach is the simplicity in finding a collision-free path with low computation times.

3.2.2 Approximate Cell Decomposition

This method has a simpler implementation. In practice, the arena space has been subdivided in a series of n equal adjacent cells which some of them may be overlapped by the obstacles. The subdivision works using the arena as an initial cell and subdividing it into n-cells using as the new cells limits the middle point of its edges and the centroid of the

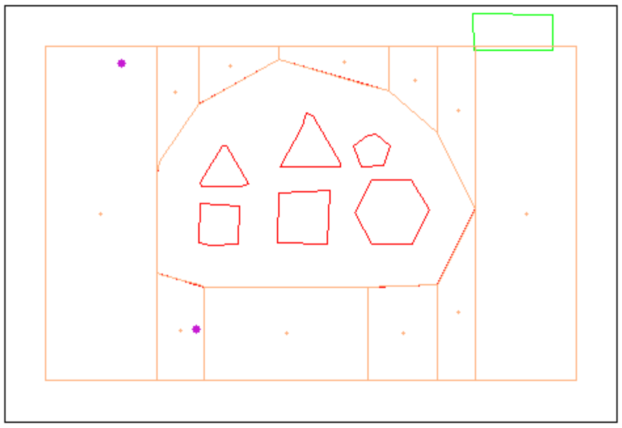

Figure 5. Example of Exact Cell Decomposition

arena. Iteratively this process may be applied for each new cells according to the desired resolution.
As the exact cell decomposition, the area of the obstacles are subtracted from the area of the cells. The main problem is that the cells are not defined using the vertex of the obstacles so it's important to tune their dimension. This process provides two issues:

- A size too large may cause the inability to find a collision-free path due to the fact that there may be concave cells.
- A size too small may cause too high computation time.

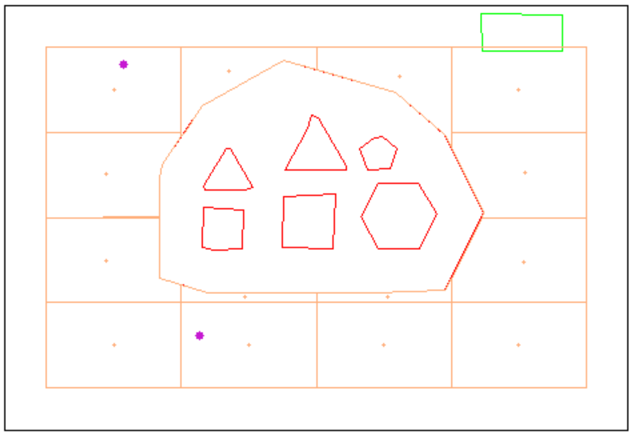

Figure 6. Example of Approximate Cell Decomposition

3.3. Cells Connections Graph

After the generation of the free space cells in which the environment have been discretized it is also necessary to detect the connections that each of them have towards all the others. This is done to account for faster access to nearby cells during the computations and also to rapidly express this knowledge in generating the files for the PDDL solver. To store the connections it has been implemented a specific data structure called *Connection map* which resents a graph structure where each element is called *Master_node*. Each Master_node is comprehensive of two lists representing adjacent and diagonal cells. Their difference consists in the number of points in which they touch the Master_node, practically diagonal cells have one point in common, while adjacent cells have more than one. Together with those lists, each element in the structure has also a pointer to the cell which represents and a variable representing the mean area in its sector, being the set of cells to which the Master_node is connected to. Whenever a new cell is added to the structure, the Connection_map automatically generate a new node and checks the connection of the cell with all the other stored elements and populates the lists of the new node using the results of this comparison.

4. Path Planning

The part of the algorithm which rules the agent behaviour has been written exploiting the PDDL formulation to employ off-the-shelf solvers like *Metric-FF* which provide an efficient solution to the fugitive-pursuer task. In particular it has not been used the vanilla version of *Metric-FF* while a newer version of it has been preferred. This version allows for metric minimization and different search algorithms like A* or BFS. The PDDL domain and problem formulation have many differences for each agent depending by the tasks that they have to carry on.

4.1. Environment in PDDL

To allow a correct formulation of the environment into a PDDL problem it has been necessary to translate the free space cells and agents into PDDL entities. An hierarchy of PDDL types has been defined to allow flexibility in the planning process. The types defined are:

- *catcher* and *fugitive* which super type is *robot*.
- *CELL* and *gate* which super type is *location*.

The PDDL actions available for each agent depends by their type: both the fugitive and the pursuer can perform the action *move* between two connected cells. In addition the pursuer has the action *capture* which determines that a fugitive agent is captured if they are in the same cell or if it is in a cell that the fugitive will have to reach in order to escape and it has not been visited yet.
The predicates available are:

- *is_in ?a ?l*: it is the predicate expressing that the agent ?a is in the location ?l.

- *connected ?l1 ?l2*: it is the predicate expressing that a location ?l1 is directly connected to another one ?l2 meaning that they touch in at least one point.

- *is_diagonal ?c1 ?c2*: it is the predicate expressing that the cells ?c1 and ?c2 are in a diagonal relationship meaning that they touch only in one point.

- *is_CELL_id ?c*: it is the predicate stating that a cell has the id "id".

- *captured ?ap ?af ?c*: it is the predicate expressing that the pursuer agent ?ap has successfully captured the fugitive agent ?af in cell ?c.

- *visitable ?c*: it is the predicate used to tell if a cell ?c is inside the plan of the fugitive agent.

- *visited ?c*: it is the predicate used to tell whether a cell ?c has been visited by the fugitive agent.

- *escaped ?af*: it tells whether a fugitive agent ?af has escaped.

Specifically, the predicates is_CELL_id and is_diagonal are exploited to define a movement cost for the agents.

4.2. Fugitive

The agent, embodying the role of the fugitive, has the task to escape from one of the available gates and express a much more complex behaviour rather than the pursuer. This because it has at its disposal three different ways to chose its final location. The available fugitive behaviours are:

1. Least steps.

2. Undeterministic.

3. Aware.

4.2.1 Least Step Behaviour

In the least step behaviour the fugitive agent will make a unique PDDL domain file and multiple problem files, one for each gate available. Then it will run the PDDL solver and once the plans are found the one having the least number of actions is retained and used as the valid one.

4.2.2 Undeterministic Behaviour

In case the fugitive agent uses an undeterministic behaviour it will just perform a random sample over the available gates ids and set the sampled one as its desired exit, then it will run the planner and save the result as the plan to follow.

4.2.3 Aware Behaviour

Lastly, if the fugitive agent behaviour is of type aware what happens is that the entity will know about the existence of the pursuer and also where it is. With these information it will try to chose the most suitable gate for its escape. The decision process works as follows:

1. The fugitive agent will compute the plans from its location to all the available gates.

2. The agent will then compute the plan from its location to the pursuer: it is supposed that it is the same plan that the pursuer will follow to capture the fugitive.

3. The fugitive will then parse the pursuer plan against its plans to the gates and whenever a cell of free space is common to both, the plan to the gate it is considerate an unsafe path and cancelled.

At the end of this processing if no plans have been left then the algorithm will rely on the least steps solution, otherwise it will check which retained plan has the minimum number of steps and saves it as the valid one.

4.3. Pursuer

The agent, embodying the role of the pursuer, has the task to catch the fugitive agent before it flees from one of the available gates. The only clue regarding the fugitive known to the pursuer is its location in a certain time step. Given the lack of informations it is almost impossible to retrieve an optimal strategy able to capture the opponent. Due to this, it has been thought to inject inside the pursuer a bit of human reasoning allowing it to suppose the behaviour that its antagonist has, which is set directly by the user. Having at its disposal a supposed fugitive's behaviour, the pursuer will generate some new instances of a fugitive, called ghosts. These new instances have the fugitive's supposed behaviour and they will provide an escape plan.
Once this plan is available, the pursuer will parse it and identifies each cell in use, to define a new PDDL type, subset of the CELL type, for each specific entity.
Then the pursuer will translate the plan into a sequence of conditional effects PDDL statements. In those the cells types that the planner can use for the cells are exclusively the generated ones and so it implicitly force the planner in using only a specific cell for that peculiar conditional effect. Thanks to this translation, incorporated with the pursuer's *move* action, it has been possible to make the pursuer aware of its own action consequences during the planning process and so exploit the planner to provide an optimal solution. Furthermore, the goodness of the result of the planning step is ensured by asking to the planner to minimize the space that the pursuer will go through. In case no valid plan has been found, due to the obstacles and

agents positioning, the catcher will go to the gate which supposes to be the same of the fugitive.
An example of plan translation is shown below.

The chunk of plan:
- 0: MOVE FUGITIVE CELL_9 CELL_11
- 1: MOVE FUGITIVE CELL_11 CELL_22

Which means "Move the fugitive from cell_9 to cell_11 and then from cell_11 to cell_22" will be translated into:

```
(when
    (is_in ?r_fugitive ?c_CELL_11 )
    (and
        ( is_in ?r_fugitive ?c_CELL_22 )
        ( not ( is_in ?r_fugitive ?c_CELL_11 )
    )
)
(when
    (is_in ?r_fugitive ?c_CELL_9 )
    (and
        ( is_in ?r_fugitive ?c_CELL_11 )
        ( not ( is_in ?r_fugitive ?c_CELL_9 )
    )
)
```

Where ?r_fugitive is the variable representing a robot of type fugitive, ?c_CELL_11, ?c_CELL_22 and ?c_CELL_9 are variables which type must respectively be of type CELL_11, CELL_22 and CELL_9. It is important to point out as the conditional effects are written in an inverse order with respect to the input plan, this because otherwise a chain effect would rise during the planning resulting in the pursuer always losing.

5. Dubins Path Computation

Once the planner has finished computing the strategy to follow, the result needs to be refined by implementing a motion planning algorithm allowing the robot to move from an initial position to a final one.
For simplicity, in the design and constraints of the robots, the speed is constant and the dubins path was calculated.

5.1. Dubins Path

A Dubins Curve is the optimal curve that satisfies these constraints:

- Has an initial state (x_0, y_0, θ_0).
- Has final configuration (x_f, y_f, θ_f).
- Has a maximum curvature angle K_{max}. This value is limited by the physical capabilities of the robot in how tight it can turn. It has been tested that it can go up to $K_{max} = 50\frac{1}{m}$ without any problems.

Each curve is composed by three arcs which can: rotate, go right, go left or go straight. Among all these combinations of curves ($LRL, RSR, LS, SR, R, S, etc\ldots$) it is possible to find the optimal one.
To calculate the multi-points Dubins path an iterative dynamic programming approach is used. This method is formulated as a minimization problem over all the points for which the Dubins maneuvers yield a corresponding total distance L, to minimize.
Let's define the $D_j(\theta_j, \theta_{j+1})$ function that returns the optimal dubins curve between two points, where θ_j is the angle of the point p_j.
The solution of the multipoint dubins problem is given by the angles that minimizes the total length L of the path:

$$L = \min_{\theta_0,\ldots,\theta_n} \sum_{j=0}^{n-1} D_j(\theta_j, \theta_{j+1})$$

This formulation suppose $\theta \in \Theta$ where $\Theta = [0, 2\pi)$ but it is possible to discretize the interval into a smaller subset. To solve this problem it is possible to use a brute-force method and compute all the possible combinations but this approach has a complexity of $O(k^n)$ where k is the number of discretized angles $\in \Theta$ and n is the number of points.
Using the iterative dynamic programming the complexity goes down to $O(nk^2)$. Let's redefine the function $L(j, \theta_j)$ as the function that gives the length of the solution of the sub-problem, from point P_j, with angle θ_j, onwards:

$$L(j, \theta_j) = \min_{\theta_{j+1},\ldots,\theta_n} D_j(\theta_j, \theta_{j+1}) + \cdots + D_{n-1}(\theta_{n-1}, \theta_n)$$

The problem can be expressed recursively:

$$L(j, \theta_j) = \min_{\theta_{j+1},\ldots,\theta_n} D_j(\theta_j, \theta_{j+1}) + L(j+1, \theta_{j+1})$$

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

To keep the computation time contained, the number o k angles in discretization is kept small and this leads to a local optimum. Another way to keep the computation time contained but exploiting an high number of angles is to return backwards only the top k-best founded paths at each iteration. Those will still lead to a local optimum and with the risk that they would finish into dead ends.

6. Observations

6.1. Exact-Cell Decomposition

Pros:

- Fast execution time.
- Convex cells. This helps a lot in the search for plausible dubins paths without any collision with the walls.

Cons:

- The number of vertical sweeps depends on the number of vertices of the obstacles. With very jagged obstacles it is possible to have a good approximation of the free space but it is also possible to have cells that are too thin, due to the presence of two vertices very close to each other. If the obstacles have few vertices, cells of free space can be obtained with very different and very large areas. This means that during the execution of a step the two robots can move for very different distances and that therefore when one has completed its execution the other one continues to move thus covering a greater distance. So at the beginning of the task the pursuer might believes he can capture the fugitive but during the first step they could travel very different distances thus making the task unfeasible. The different cell's size is observable in Fig. 5.
- In the extreme case without the presence of any obstacle there is no decomposition and therefore the robots find themselves having to move in a single cell, the shrunk arena itself.

6.2. Approximate-Cell Decomposition

Pros:

- Ease of implementation.
- The cells are more or less regular and therefore the space covered from one step to another is the same.
- There is much more freedom of planning routes.
- In the limit case without any obstacle, the free space is well sectioned and the robots have equal chances of winning.

Cons:

- Execution time increases as resolution increases due to the number of cells to be created.
- Possibility of the presence of concave cells. These make the finding the dubins path much more difficult as it is easier to collide with the walls, like highlighted in Fig. 7.
- In proximity of the obstacles it is possible that these take up a large space from the cells, leaving them with tiny areas. The movement space is thus very close to the wall making it very difficult to find a plausible dubins trajectory. This case is observable in Fig. 6.

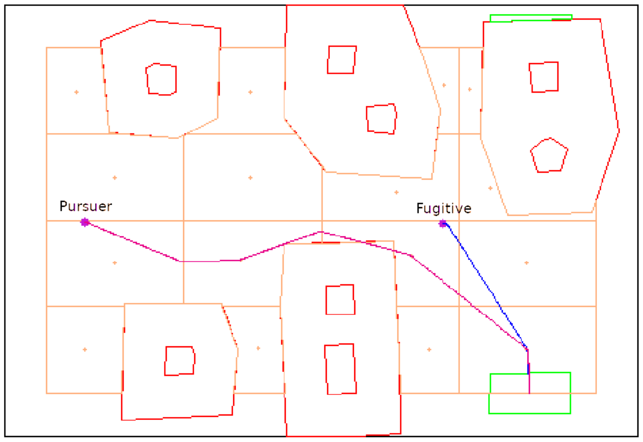

Figure 7. Limitation Approximate Cell Decomposition: low resolution.

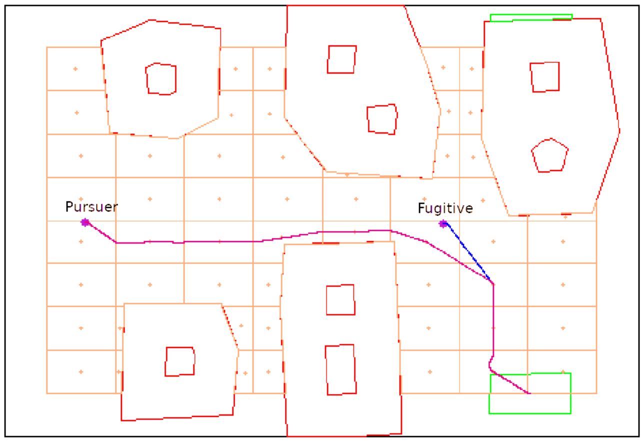

Figure 8. Limitation Approximate Cell Decomposition: high resolution.

7. Results

In this work a pursuer-evader scenario is implemented. To achieve it it was necessary to divide the work into 3 subtasks: roadmap generation, planning path, multi-points dubins path computation.
Two different methods were implemented and compared for the generation of the roadmap: the exact cell decomposition and the approximate cell decomposition. The choice of which to use may vary depending on the composition of the environment and the required execution times.
For path planning the adjacencies between the various cells have been encoded in PDDL so that Metric-FF can be used to obtain a plan to follow.
Once this is done, the computation of the multi-points dubins path has been executed using a recursive dynamic programming approach that searches for optimal local solutions.

The results obtained are satisfactory and the execution of the algorithm as well, even if it varies according to the number of cells that a robot has to pass through and the number of free space cells that must be created if the approximate-cell decomposition method is used.

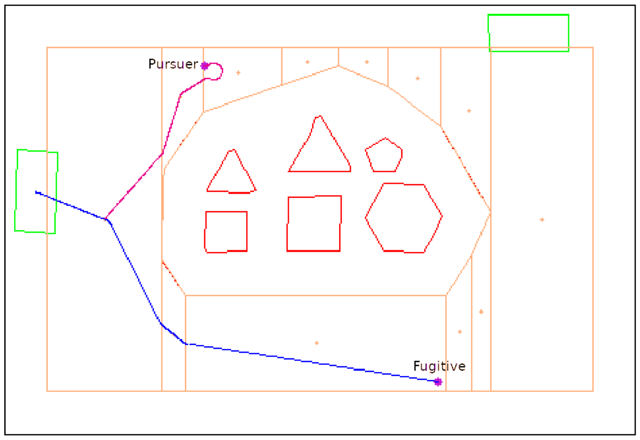

Figure 9. Dubins path in an arena decomposed with the Exact Cell Decomposition.

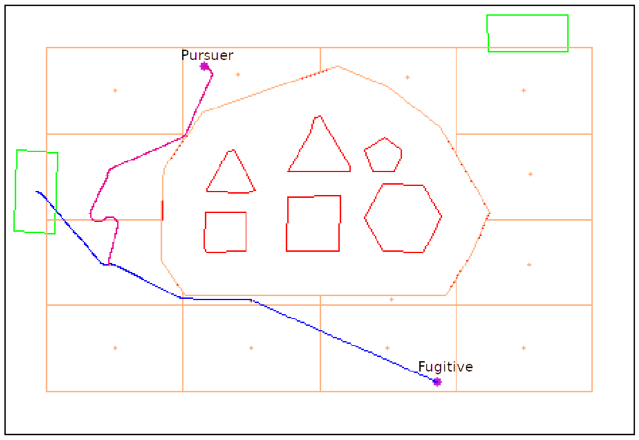

Figure 10. Dubins paths in an arena decomposed with the Approximate Cell Decomposition.

8. Future Directions

In future updates the approximate cell decomposition may be improved by creating the free-space cells using an adaptive resolution. So during their creation low resolution is used and then it is raised nearby the obstacles. In this way, few and large cells are obtained in free areas of empty space and many but small ones near the obstacles, thus increasing the precision of movement and at the same time discouraging the planner to choose a path that passes close to the obstacles. Other methods to generate the roadmap may be implemented and tested, like the voronoi diagrams or the rrt*.

Subsequently the planner may be improved in managing a collaborative environment.

The last update that should be implemented is to refine the multi-points dubins path computed, recalculating it using a small subset around the optimal angles found and approaching a global optimum.