A.M.R. Project report

Caterina Lacerra, Jary Pomponi

July 11, 2017

1 Introduction

In this project we have studied the problem of motion planning. More precisely we have used and compared different algorithms.

Informally speaking, the motion planning problem aims to find a path, for a specified robot, from a point A to a point B avoiding obstacles. In practice this problem is very hard from a computational complexity point of view, since it needs to map the environment in the configuration space of the robot.

The strength of the algorithms that we have used is that, instead of building and using an explicit representation for the robot in its configuration space, they rely on a collision checking module. With this approach we build a graph of feasible trajectories in the obstacle free space. Then the road map can be used to build the solution to the planning problem for a specific robot.

2 Motion planning problem

In this section we will define the planning problem and the primitive procedures that will be used by the algorithms, then we will explain them.

2.1 Problem formulation

In this section we will formalize the problem of path planning.

Let $\chi \in (0,1)^d$ be the configuration space, where $d \in \mathbb{N}$ with $d \geq 2$, and χ_{obs} is the obstacle region of the space. The χ_{free} is the portion of the space that does not contains obstacles. Then we define an initial position $x_{init} \in \chi_{free}$ and a goal region $\chi_{goal} \subset \chi_{free}$. A path planing problem is defined by a triple $(\chi_{free}, x_{init}, \chi_{goal})$.

The goal of a path planner is to find a feasible path from x_{init} to χ_{goal} in χ_{free}, if it exists, and report failure otherwise.

To do this we build a DAG (direct acyclic graph) of possible road maps from the initial point to the other point in χ_{free}. Then we get the optimal path from all the possible ones.

2.2 Primitive procedures

Before discussing the implemented algorithm we introduce the principal procedures that we will use.

SampleFree: is a function that returns a point $x_r \in \chi_{free}$. The samples are assumed to be drawn from a uniform distribution, and each one is independent from others.

NearestNode: Given a a graph $G = (V, E)$ and a point $x \in \chi_{free}$, where $v \subset \chi$, the function returns a point $x_{near} \in V$ that is the closest to x given the euclidean distance. Formally:

$$getNearest(G, x) := argmin_{v \in V} ||x - v||$$

The are other two versions of the methods. One version of this function, taken another value r, will return a set of $x \in V$ that are contained in a ball of radius r centred in x, the other one returns k nearest vertex to x.

Steer: Given two point $x, y \in \chi$, the function steer returns a point $z \in \chi$ that minimizes the distance from z to y while $||z - x|| \leq \eta$, with $\eta > 0$.

collisionFree: Given two points $x, y \in \chi_{free}$ the function return true if the line that connects x to y lies in χ_{free}.

Parent: Given a Graph $G = (V, E)$ and an $x \in V$, the method return a point $y \in V$ that satisfies $(y, x) \in E$. By convention, if x is the root node of V this function will return x.

Line: Given two points $x, y \in \chi$, the method returns a line that goes from the point x to the point y.

Cost: Given a graph $G = (V, E)$, a node $v_{start} \in V$ and another node $v_{goal} \in V$, the method will return the cost $c \in \mathbb{R}$ of the unique path from v_{start} to the node v. By convention the cost of the root node is zero. If only one node is given, the method returns the distance from the root of the graph to v_{goal}.

2.3 Existing Algorithms

Rapidly-exploring Random Trees (RRT) based algorithms are very effective for solving the path planning problem for high complex configuration space. This because that class of algorithm combines random sampling of the configuration space with biased sampling around a desired goal. A good property that follows from the random sampling is that the construction of the construction of the tree is biased towards unexplored areas of the configuration space.

This class of algorithms aim to build a road-map graph in the configuration space from a root tree to a feasible goal region. Another property of this class is that the algorithms are single query, so each road-map is relative to a single problem.

While RRTs have been shown to be extremely effective at generating feasible solution in the configuration space, they provide no control on the quality of the found path. This problem is accentuated if the space contains non uniform cost sub-spaces, And for a robot in a real world it is important to find a good path in a fast way.

The basic behaviour of this class of algorithms is the following one: at each iteration the algorithm samples a new point in the free space. Then it computes the nearest vertex in V to the sampled point, and then the algorithm tries to connect the nearest vertex to this random point, using the **Steer** function. If the connection does not collide with any obstacle, the new vertex and the edge that connects the new point and the nearest one, are added to the graph. This is shown in figure 1.

The algorithms in this section are probabilistically complete. This means that, if a path exist from the initial vertex to the goal one, the probability that the algorithm fails to find that path asymptotically approaches zero with $n \to \inf$.

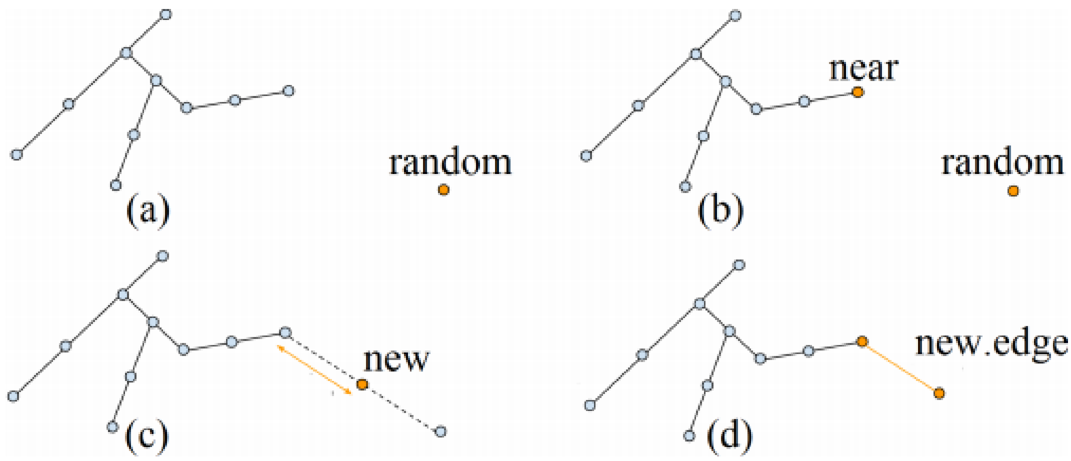


Figure 1: The extending procedure

2.3.1 Rapidly-exploring Random Trees (RRT)

This is the simplest RRT algorithm. In this basic version, the algorithm does not take into account the length of the path, if one is found, from the start vertex to the goal region. On the other hand it is very fast in exploring, randomly, the configuration space.

In this version of the algorithm we perform the basic operations n times, so the algorithm does not stop it reaches the goal but it keeps running until the number of iteration is not reached.

Algorithm 1: RRT

G = $(x_{init}, \emptyset)$
for i = 1... n do
    $x_{rand} \leftarrow$ SampleFree
    $x_{nearest} \leftarrow$ NearestNode(G, x_{rand})
    $x_{new} \leftarrow Steer(x_{nearest}, x_{rand})$

```
    if collisionFree(x_nearest, x_new) then
        V ← V ∪ {x_new}
        E ← E ∪ {(x_nearest, x_new)}
```

One negative result tell us that the cost of the best path found by the RRT procedure converges to a random variable. This implies that the cost of the solution converges to a suboptimal value, with probability one. It is possible to implements different heuristics in order to improve the solution, but the the sub-optimality still remains. So this basic algorithm is probabilistically complete but not asymptotically optimal.

2.3.2 Anytime RRT

The main idea behind this approach is that we have a limited amount of time. In this time we run a basic RRT algorithm, and if there is some time left, it tries to generate a new RRT path ensuring that the cost of that new path is lower than the previously found. This is achieved by limiting the nodes added to the tree to the only ones that can contribute to a solution with lower overall cost.

This approach ensures that each new tree contains a path that cost less with respect to all the previously built trees. However this approach does not guarantee that a new solution can be produced. To ensure that the new solution will cost less than the previous one the algorithm incorporate cost consideration and bias factor on the methods.

Algorithm 2: Anytime RRT

```
Main()
 G = (x_start, ∅)
 d_b ← 1
 c_b ← 0
 c_s ← inf
 loop
  G = (x_start, ∅)
  c_n ← growRRT(G)
  if(c_n ≠ null) then
   postCurrentSolution(G)
   c_s ← (1 − ε) * c_n
   d_b ← d_b − δ_d
   if (db_<0) then
        d_b ← 0
   c_b = c_b + δ_d
   if (c_b > 1) then
        c_b ← 1
chooseTarget(G)
 rand ← randomRealIn[0,1]
 if (rand > goalBias)
  return x_goal
 else
  x_new ← RandomFreeConfig()
  attemp ← 0
  while (cost(x_start, x_new)
     + cost(x_new, x_goal) > c_s ) do
   q_new ← RandomFreeConfig()
   attemp ← attemp+1
   if(attemp > maxAttemp) then
    return null
  return q_{new}
```

Algorithm 3: Anytime RRT

```
GrowRRT()
 x_new ← x_start
 time ← 0
 while(||x_new − x_goal||> Threshold) do
  x_target ← chooseTarget(G, x_target)
  if (x_target ≠ null) then
   x_new ← extendToTarget(G, x_target)
   if(x_new ≠ null) then
    V ← V ∪ {x_new}
    updateTime(time)
    if (time > maxTimePerRRT) then
     return null
  return \textbf{distance}(x_start, x_new)
extendToTarget(G, x_target)
 K_near ← kNearestNeighbors(G, x_target, k)
 while K_near is no empty do
  remove x_tree with minimum
     selCost(G, x_tree, x_target) from K_near
  Q_ext ← generalExtensions(x_tree, x_target)
  x_new ← argmin_{q∈Q_ext} ||x_tree − x||
  c ← cost(x_start, x_tree) + ||x_tree − x_new||
  if c + cost(x_new, x_goal) < c_s then
   return x_new
 return null
SelCost(G, x, x_target)
 return d_b * distance(x, x_target)+
    c_b * cost(x_start, x)
```

2.3.3 RRT*

This algorithm still uses a tree graph, but with the improvement that the graph is modified at each iteration, if the algorithm discover a better path from the root to the actual node.

This algorithm adds a point to the graph in the same way as RTT does, but it also considers connection from a subset of the vertex called xNear. This set contains the nodes that are inside a ball of radius

$$min\left\{\gamma_{RRT}\left(\frac{log(card(V)}{card(V)}\right)^{1/d}, \eta\right\}$$

where d is the dimension of the configuration space χ, η is the constant of steering function and γ_{RRT} is a constant related related to the shape of the configuration space, and it is equal to

$$2\left(1+\frac{1}{d}\right)^{1/d}\left(\frac{\mu(\chi_{free})}{\varsigma_d}\right)^{1/d}$$

where $\mu(\chi_{free})$ denotes the volume of the free space and ς_d is the volume of the unit ball in the d-dimensional Euclidean space.

In our case d is equal to 2.

Algorithm 4: RRT*

G = $(x_{init}, \emptyset)$
for i = 1... n do
  $x_{rand} \leftarrow$ **SampleFree**
  $x_{nearest} \leftarrow$ **NearestNode**(G, x_{rand})
  $x_{new} \leftarrow$ **Steer**$(x_{nearest}, x_{rand})$
    if **collisionFree**$(x_{nearest}, x_{new})$ then
      $radius \leftarrow min\{\gamma_{RRT}(\frac{log(card(V)}{card(V)})^{1/d}, \eta\}$
      $xNear \leftarrow$ **nearestNode**$(G, x_{new}, radius)$
      $V \leftarrow V \cup \{x_{new}\}$
      $x_{min} \leftarrow x_{nearest}$
      $c_{min} \leftarrow$ **Cost**$(x_{nearest}) + ||x_{nearest} - x_{new}||$
      foreach $x_{near} \in xNear$ do
        if **collisionFree**$(x_{near}, x_{new}) \wedge$ **Cost**$(x_{near}) + ||x_{near} - x_{new}|| < c_{min}$
        then
          $xmin \leftarrow x_{near}$
          $c_{min} \leftarrow$ **Cost**$(x_{near}) + ||x_{near} - x_{new}||$
      $E \leftarrow E \cup \{(x_{min}, x_{new})\}$
      foreach $x_{near} \in xNear$ do
        if **collisionFree**$(x_{near}, x_{new}) \wedge Cost(x_{near}) + ||x_{near} - x_{new}|| <$ **Cost**(x_{near})
        then
          $x_{parent} \leftarrow$ **Parent**(x_{near})
          $E \leftarrow (E \setminus \{(x_{parent}, x_{near})\} \cup \{(x_{new}, x_{near})\})$

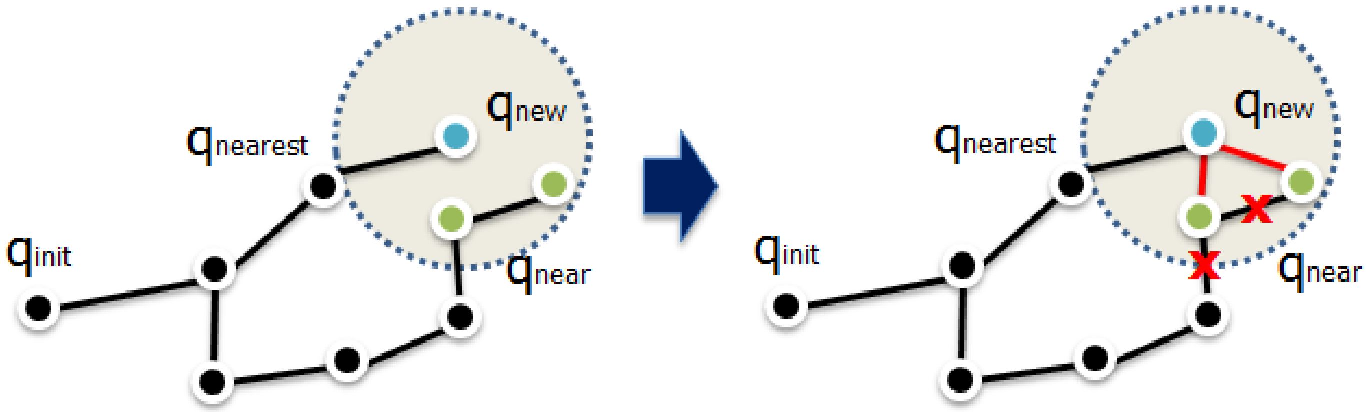


Figure 2: The rewriting process

The basic iteration is the same of RRT. The main difference is that, if the connection between $x_{nearest}$ and x_{new} is collision free, the algorithm takes all the configurations inside the ball centred in x_{new}. Then, we get the configuration in the sphere that minimizes the cost of the path from x_{init} to that configuration, passing trough x_{new}, and connects that configuration to x_{new}.

After that the algorithm rewrites the tree. For each configuration inside the sphere the algorithm check if is convenient to change the father of that configuration with x_{new}. This is true if the cost of the path will be lower passing trough x_{new}.

The rewriting process is shown in figure: 2.

3 Set up and Experiments

In this section we will explain the implementation and we'll show the results of the experiments.

3.1 Experiments

The first experiment aims to show the difference between RTT and RRT*. Both algorithms were run with the same sample sequence and without goal bias, so the differences resides in the edges of the graph. The results are shown in figure: 3. We can see that, despite we don't have a goal bias, RRT* can decrease the path cost by working on local edges, even without a goal bias.

The second set of experiments involves a space containing obstacles. In this case we run each algorithm for 6000 iterations with a goal bias equals to 0.7.

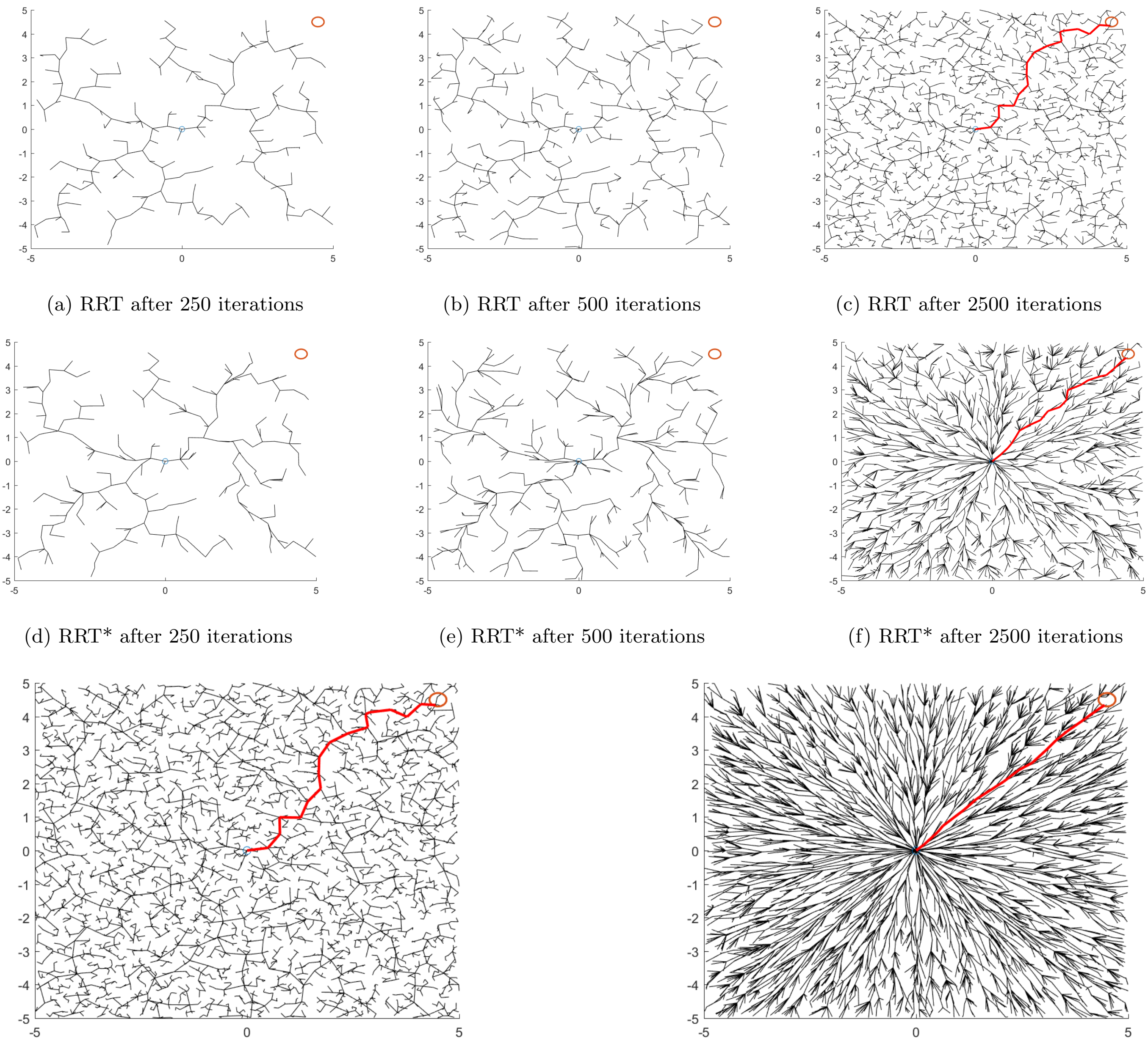
In the first scenario we have a narrow passage and a wide one. We see that both of the algorithms, RRT and RRT*, can go through the narrow passage. We can also see that, since RRT* improve the cost locally to the sampled configuration, it is more difficult to improve the passage in the narrow space. For this reason the difference in cost between the two paths found is not notable. The second scenario differs from the first one for the lack of the narrow passage. The images show that the RRT* can improve the path significantly respect the RRT version.

The third scenario is a simple and without relevant properties, while in the last one we have the starting point "trapped". In both cases the RRT* works better than the basic version.

All of the scenarios are shown in figure: 4.

The last experiment aims to show the performances of the RRT* and Anytime RRT compared. We have an obstacle free space that contains some areas that modify the cost of the edges. The darkest one double the cost of the edge that lies in the area, while the brighter will half the cost. In this experiments makes no sense compare the basic RRT. The resulting plot are shown in figure: 5.

The result for the set of experiments containing objects are show in figure 6 and 7. As we expected the cost of the path found by RRT is constant, because the algorithm traps itself and there is no chance to discover a better path. On the other hand the path found by RRT* will become better at each iteration, with an improvement rate that depends on the problem instances.



(a) RRT after 250 iterations

(b) RRT after 500 iterations

(c) RRT after 2500 iterations

(d) RRT* after 250 iterations

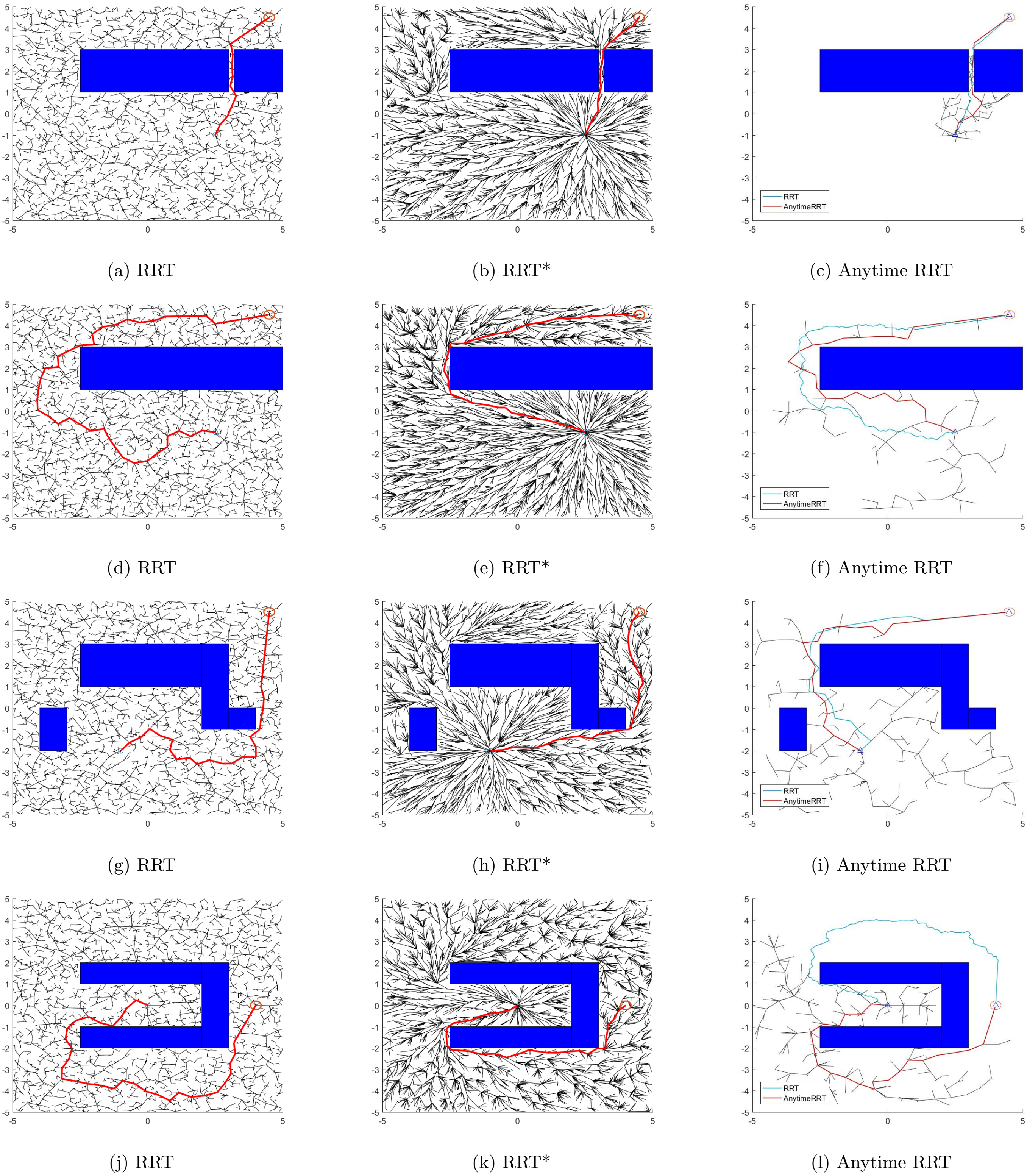
(e) RRT* after 500 iterations

(f) RRT* after 2500 iterations

(g) RRT after 5000 iterations

(h) RRTS* after 5000 iterations

Figure 3: The configuration exploration differences between RRT and RRT* in an empty space, without goal bias.



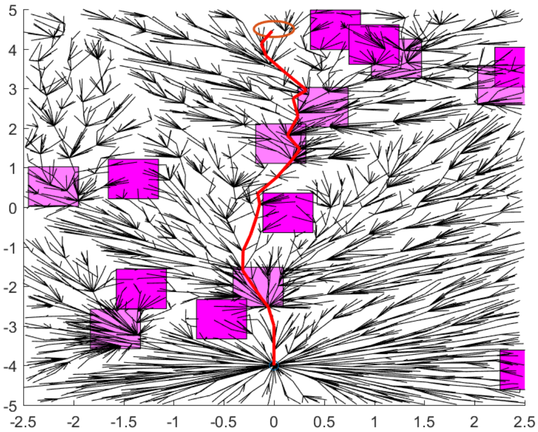
(a) RRT (b) RRT* (c) Anytime RRT

(d) RRT (e) RRT* (f) Anytime RRT

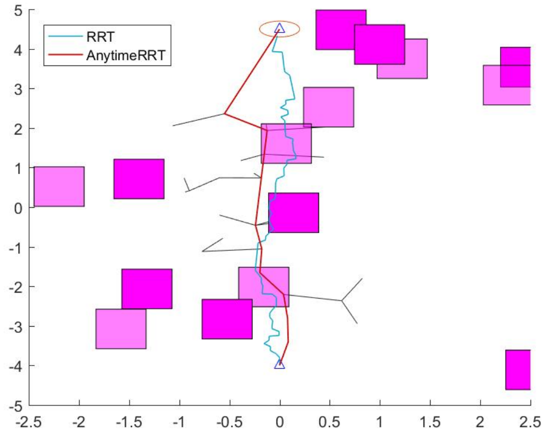
(g) RRT (h) RRT* (i) Anytime RRT

(j) RRT (k) RRT* (l) Anytime RRT

Figure 4: Some tests done in different obstacle environment. RRT and RRT* were run for 6000 iterations.

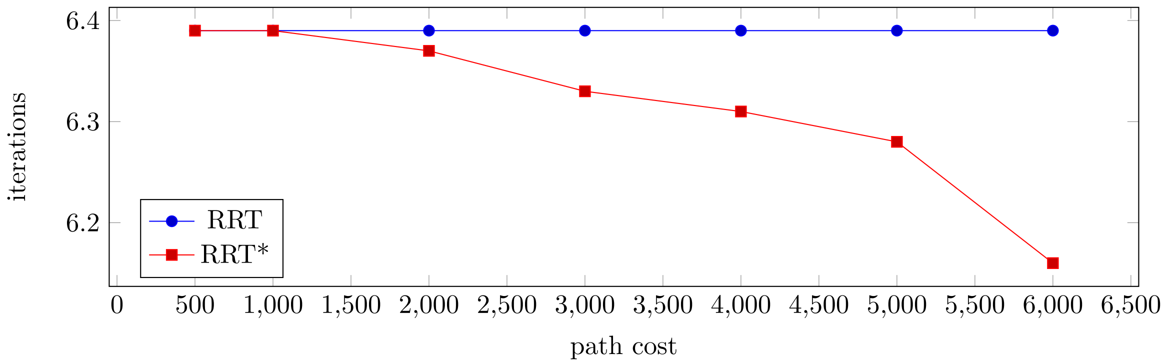


(a) RRT*

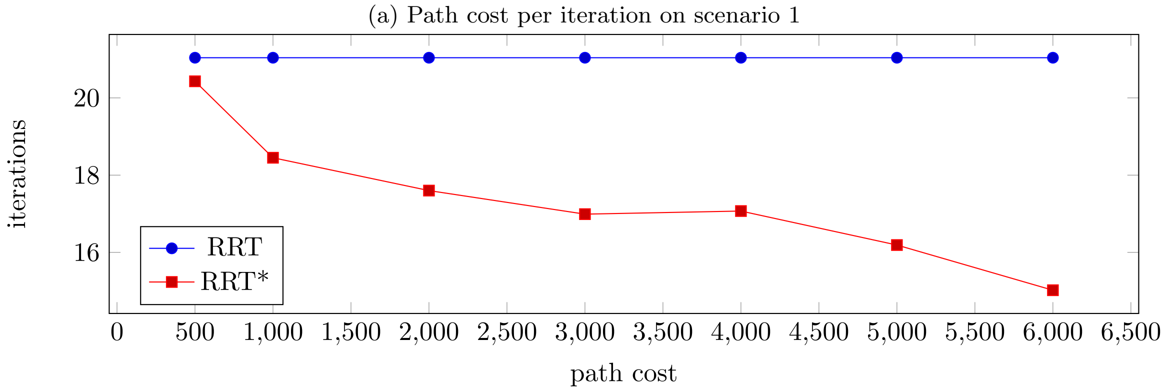


(b) Anytime RRT

Figure 5: Results in an environment contains non uniforms cost area

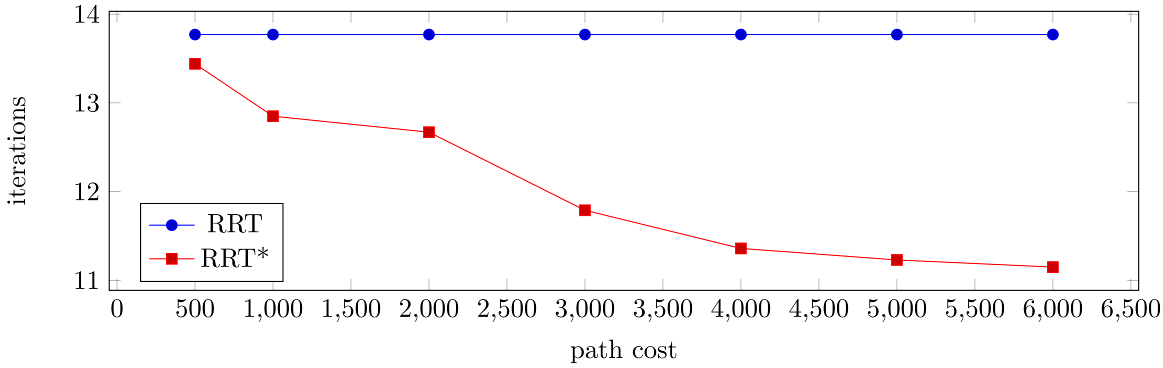


(a) Path cost per iteration on scenario 1

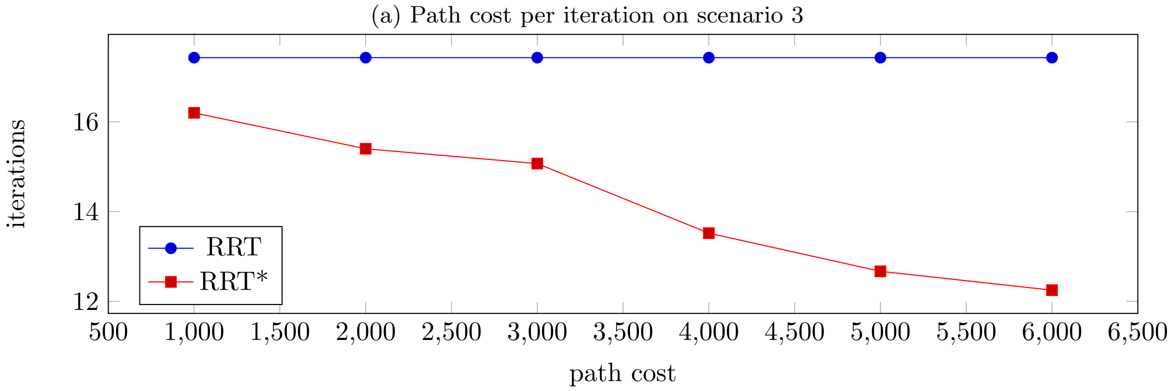


(b) Path cost per iteration on scenario 2

Figure 6: Results for the fist two scenarios



(a) Path cost per iteration on scenario 3



(b) Path cost per iteration on scenario 4

Figure 7: Results for the last two scenarios

4 Conclusions

Summarizing, the aim of our work was to implement and compare the performances of the RRT algorithm with those of the RRT$*$ and of the Anytime RRT. From a theoretical point of view, all the algorithms described in this paper are probabilistically complete, but while the RRT tries to randomly explore all the free space, the other two approaches are oriented to find an optimal solution, respectively by rewiring the graph when a better solution is found or by adding to the graph only nodes that can lead to a smaller cost.
This, as we have seen with the experiments, leads the RRT* and the Anytime RRT to outperform the RRT, finding shortest paths, or paths through areas with a lower cost.