

A.M.R. Project report

Caterina Lacerra, Jary Pomponi

July 7, 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 for a computational complexity point of view, since we need to map the environment in the configuration space of the robot.

The strength of the algorithm 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. The road map can be used to build the solution for the planning problem for a specific robot.

2 Motion planning problem

In this section we will define the planning problem and the primitive procedure that will be used by the algorithms, then we will expose 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 contain obstacles. We define an initial position $x_{init} \in \chi_{free}$ and a goal region $\chi_{goal} \subset \chi_{free}$. A path planning 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 return a point $x_r \in \chi_{free}$. The samples are assumed to be draw 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 $v \in \chi$, the function return a point $x_{near} \in V$ that is the closest to x given the euclidean distance. Formally:

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

There are other two versions of the method. 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 return k nearest vertex to x .

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

collisionFree: Given two point $x, y \in \chi_{free}$ the function return true if the line that connect 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 satisfy $(y, x) \in E$. By convention, if x is the root node of V this function will return x .

Line: Given two points $x, y \in \chi$, the methods return a line that go 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 methods 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 give, the method return the distance from the root of the graph to v_{goal} .

2.3 Existing Algorithm

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

This class of algorithm 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 show 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 is important to find a good path in a fast way.

The basic behaviour of this class of algorithms are the following one: at each iteration the algorithm sample a new point in the free space. Then calculate 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 obstacles, the new vertex and the edge, that connects the new point and the nearest one, are added to the graph. This is show in *figure 1*.

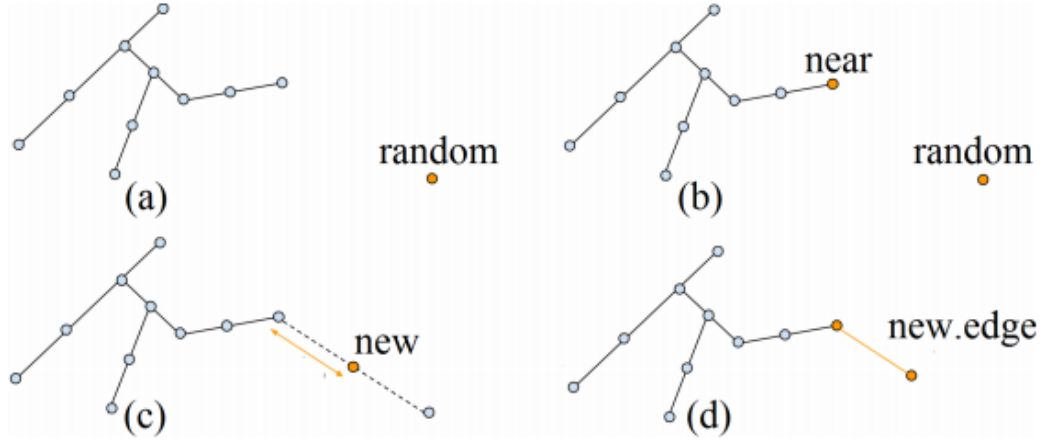


Figure 1: The extending procedure

In this section we will see the existing algorithm that we have implemented and compared.

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 distance of the path, if one is found, from the start vertex to the goal region. On the other 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 if reach the goal but keep running until the number of iteration is not reached.

Algorithm 1: RRT

```

 $G = (x_{init}, \emptyset)$ 
for  $i = 1 \dots n$  do
     $x_{rand} \leftarrow \text{SampleFree}$ 
     $x_{nearest} \leftarrow \text{NearestNode}(G, x_{rand})$ 
     $x_{new} \leftarrow \text{Steer}(x_{nearest}, x_{rand})$ 
    if  $\text{collisionFree}(x_{nearest}, x_{new})$  then
         $V \leftarrow V \cup \{x_{new}\}$ 
         $E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$ 

```

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 than 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 assures that each new tree contains a path that cost less regard all the previously trees. However this approach does not guarantees that a new solution can be produced. To

ensure that the new solution will cost less then the previously one the algorithm incorporate cost consideration and bias factor on the methods.

Algorithm 2: Anytime RRT

```

Main()
   $G = (x_{start}, \emptyset)$ 
   $d_b \leftarrow 1$ 
   $c_b \leftarrow 0$ 
   $c_s \leftarrow \text{inf}$ 
  loop
     $G = (x_{start}, \emptyset)$ 
     $c_n \leftarrow \text{growRRT}(G)$ 
    if ( $c_n \neq \text{null}$ ) then
      postCurrentSolution( $G$ )
       $c_s \leftarrow (1 - \epsilon) * c_n$ 
       $d_b \leftarrow d_b - \delta_d$ 
      if ( $d_b < 0$ ) then
         $d_b \leftarrow 0$ 
       $c_b = c_b + \delta_d$ 
      if ( $c_b > 1$ ) then
         $c_b \leftarrow 1$ 
  chooseTarget( $G$ )
   $\text{rand} \leftarrow \text{randomRealIn}[0, 1]$ 
  if ( $\text{rand} > \text{goalBias}$ )
    return  $x_{goal}$ 
  else
     $x_{new} \leftarrow \text{RandomFreeConfig}()$ 
     $\text{attemp} \leftarrow 0$ 
    while ( $\text{cost}(x_{start}, x_{new})$ 
      +  $\text{cost}(x_{new}, x_{goal}) > c_s$ ) do
       $q_{new} \leftarrow \text{RandomFreeConfig}()$ 
       $\text{attemp} \leftarrow \text{attemp} + 1$ 
      if ( $\text{attemp} > \text{maxAttemp}$ ) then
        return null
    return  $q_{new}$ 

```

Algorithm 3: Anytime RRT

```

GrowRRT()
   $x_{new} \leftarrow x_{start}$ 
   $\text{time} \leftarrow 0$ 
  while ( $\|x_{new} - x_{goal}\| > \text{Threshold}$ ) do
     $x_{target} \leftarrow \text{chooseTarget}(G, x_{target})$ 
    if ( $x_{target} \neq \text{null}$ ) then
       $x_{new} \leftarrow \text{extendToTarget}(G, x_{target})$ 
      if ( $x_{new} \neq \text{null}$ ) then
         $V \leftarrow V \cup \{x_{new}\}$ 
        updateTime( $\text{time}$ )
        if ( $\text{time} > \text{maxTimePerRRT}$ ) then
          return null
    return  $\text{distance}(x_{start}, x_{new})$ 
extendToTarget( $G, x_{target}$ )
   $K_{near} \leftarrow \text{kNearestNeighbors}(G, x_{target}, k)$ 
  while  $K_{near}$  is no empty do
    remove  $x_{tree}$  with minimum
       $\text{selCost}(G, x_{tree}, x_{target})$  from  $K_{near}$ 
     $Q_{ext} \leftarrow \text{generalExtensions}(x_{tree}, x_{target})$ 
     $x_{new} \leftarrow \text{argmin}_{q \in Q_{ext}} \|x_{tree} - x\|$ 
     $c \leftarrow \text{cost}(x_{start}, x_{tree}) + \|x_{tree} - x_{new}\|$ 
    if  $c + \text{cost}(x_{new}, x_{goal}) < c_s$  then
      return  $x_{new}$ 
  return null
SelCost( $G, x, x_{target}$ )
  return  $d_b * \text{distance}(x, x_{target}) +$ 
     $c_b * \text{cost}(x_{start}, x)$ 

```

2.3.3 RRT*

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

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

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

where d are the dimension of the configuration space χ , η is the constant of steering function and γ_{RRT} are 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 = (xinit, ∅)
for i = 1... n do
  xrand ← SampleFree
  xnearest ← NearestNode(G, xrand)
  xnew ← Steer(xnearest, xrand)
  if collisionFree(xnearest, xnew) then
    radius ← min{γRRT( $\frac{\log(\text{card}(V))}{\text{card}(V)}$ )1/d, η}
    xNear ← nearestNode(G, xnew, radius)
    V ← V ∪ {xnew}
    xmin ← xnearest
    cmin ← Cost(xnearest) + ||xnearest - xnew||
    foreach xnear ∈ xNear do
      if collisionFree(xnear, xnew) ∧ Cost(xnear) + ||xnear - xnew|| < cmin
      then
        xmin ← xnear
        cmin ← Cost(xnear) + ||xnear - xnew||
    E ← E ∪ {(xmin, xnew)}
    foreach xnear ∈ xNear do
      if collisionFree(xnear, xnew) ∧ Cost(xnear) + ||xnear - xnew|| < Cost(xnear)
      then
        xparent ← Parent(xnear)
        E ← (E \ {(xparent, xnear)}) ∪ {(xnew, xnear)}

```

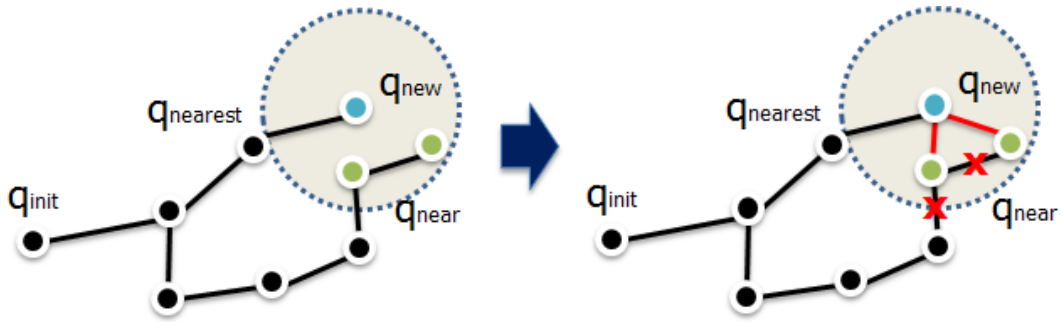


Figure 2: The rewrite process

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

After that the algorithm rewrite 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 rewrite process is show 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 Implementation of the algorithm

3.2 Experiments

4 Conclusion