Robot Autonomy (16-662, S13) Lecture #12 (Wednesday Feb 27, 2013)

# Sampling-Based Methods

*Lecture by: Siddhartha Srinivasa* *Scribing by: Yijie Wang, Ran Wang*

## 1. Introduction on Sampling-Based Methods

This scribing discusses sampling-based motion planning methods. Comparing with methods that construct boundary representations of configuration space obstacles, sampling-based methods use only information from a collision detector when they search the configuration space.

Figure 1 shows the typical sequence of sampling-based methods. Given a geometric model, an algorithm keeps coming up with a potential configuration and performs collision checking to verify the validity of the configuration. The process ends when a configuration matches the goal configuration. In any sampling-based method, normally two questions are asked:

1. When and where in the configuration space should the algorithm sample?
2. How should the algorithm search the configuration space?

Since collision checking is performed when necessary, the algorithm does not know where the object is in configuration space . Instead, its knowledge about the object in configuration space looks like what is shown in Figure 2.

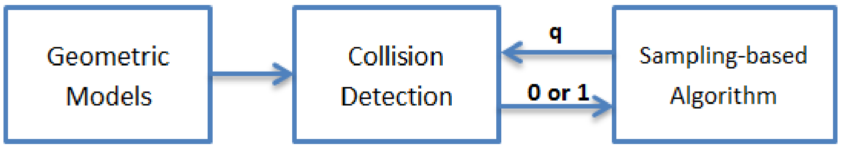

Figure 1 Typical Sequence of Sampling-Based Methods

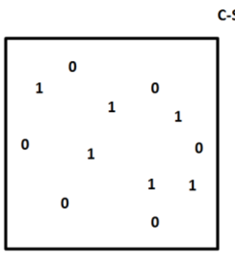

Figure 2 Collisino Checking Result

Sampling-based Methods can be divided into two categories: multi-query methods and single-query methods. Table 1 shows the comparison between these two categories.

Table 1. Comparison between multi-query methods and single-query methods

| | **Multi-query** | **Single-query** |
|---|---|---|
| Phases | 1. Roadmap construction<br>2. Searching | Roadmap construction and searching online |
| Typical algorithm | Probabilistic Roadmap (PRM) | Rapidly Exploring Random Tree (RRT) |
| Pros | Fast searching | No preprocessing |
| Cons | Inability to deal with environment changes | No memory |

For multi-query methods, a probabilistic roadmap is built during the preprocessing phase and stored inside the robot. After given the a start and a goal configuration, the robot will search the roadmap for a path joining the two nodes. As the name suggests, once the roadmap is ready, the robot can query it multiple times to plan a path connecting any two nodes. The biggest advantage for multi-query methods is that searching is really fast. However, this kind of methods works only when the environment does not change too much.

For single-query methods, no roadmaps is built ahead of the task. The robot constructs the roadmap online during the searching, with sampling conducted in the process to avoid being stuck in the local minima due to any deterministic approach. This type of methods is capable of dealing with dynamic environment, but it has no memory of past experience.

## 2. Probabilistic Roadmap (PRM)

### 2.1 Basic Concept

Probabilistic Roadmap methods proceed in two phases:

1. Preprocessing Phase – to construct the roadmap $G$
2. Query Phase – to search given $q_{init}$ and $q_{goal}$

In the preprocessing phase, a probabilistic roadmap is constructed for a given scene. The roadmap is an undirected graph $G = (N, E)$. The nodes in N are a set of configurations of the robot chosen over the free C-space. The edges in E correspond to feasible straight-line paths. For example, an edge $(a, b)$ corresponds to a feasible straight-line path connecting the configurations $a$ and $b$.

In the query phase, given a starting configuration $q_{init}$ and a goal configuration $q_{goal}$, the algorithm will search the roadmap for a path joining these two points together.

In this scribing, we mainly talk about the algorithm for preprocessing phase, i.e. how to build a roadmap.

## 2.2 Build-Roadmap Algorithm

As stated above, the goal of this algorithm is to create a probabilistic roadmap. The figure below shows the pseudo-code of the Build-Roadmap algorithm:

```
Build-Roadmap:
G.init() ; i = 0
While i < N
    $\alpha \leftarrow$ SAMPLE
    if $\alpha \in C_{free}$
        G.add Vertex($\alpha$); i = i+1
        for each $q \in ngd(\alpha, G)$
            if (connect($\alpha$, G))
                G.addEdge($\alpha$, q)
```

Figure 3 Build-Roadmap Algorithm

It starts with an empty graph: *G*(Vertex, Edge). Assume that we are going to generate *N* random points. While *i* is less than N, a sample configuration is generated and assigned to $\alpha$. Then we run the collision detector to find out whether $\alpha$ is in collision. If $\alpha$ is collision-free, add $\alpha$ to the graph *G*. After that, we need to find out all the edges that $\alpha$ is able to connect to. Define ngd($\alpha$,*G*) as the set of candidate neighbors which is chosen from all vertexes inside *G*. This set is made of nodes within a certain distance of *R*. Figure 4 illustrates the definition of ngd($\alpha$,*G*). For each point in the set ngd($\alpha$,*G*), which is one of the points in the blue area in Figure 4, we try to connect it to $\alpha$ by using a straight line if it is not already graph-connected to $\alpha$. If there is no obstacle in the configuration space within the straight line, we add the edge to the graph *G*. Otherwise, we just throw the edge away. Then we keep going until we go through all the *N* sample points.

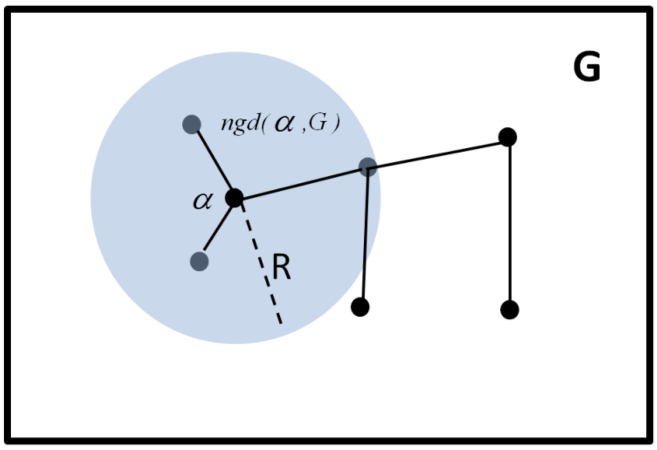

Figure 4 Definition of ngd( $\alpha$ ,G)

Notes:

1. The sample configuration can be generated either randomly or using other “clever” methods. The biggest advantage for random sampling is that there is no bias.

2. The distance of neighbor *R* is the number that tells you the scope of how far you think you can connect to configurations by straight lines. There is a tradeoff when choosing *R*. If *R* is small, neighbor points should be close to $\alpha$, which is easier to connect but takes a lot of time before filling up the whole search space. If *R* is large, $\alpha$ is able to connect far away points with straight lines, but it will be easier to fail. The algorithm says the $\alpha$ should be small. The idea is that if you start building the graph and every point is very close to each other, eventually it will be very easy to chain all the points together to go from any point to any goal.

3. How can we choose the right *R* value? Typically, what people do is to pick up some *R* values and check how many neighbors $\alpha$ is able to connect. If the number is really large, we should take a larger alpha value.

4. The advantage for PRM: it works for any $q_{init}$ and $q_{goal}$, but it spends a lot of time on preprocessing.

5. About the number of samples *N*: when the sample points are sparsely scattered, i.e. *N* is small, the algorithm will need fewer collision checks which will reduce the program running time. However, in most of the time, it might lead to suboptimal path plans. In the worst case, there might be even no path plans generated. Can we produce an algorithm that is not only dense but also good with collision check? We can use Lazy-PRM which will be discussed in Section 2.4.

## 2.3 Notes on Neighborhood

Selecting the set of neighbors to which we try to make connections is one of the most important tradeoff we need to make. On one hand, since each collision check for a connection is expensive we should try to avoid these as much as possible. On the other hand, if we try too few connections we will fail to connect the graph. Connected components in the graph play a crucial role here. We like to connect such components into larger components whenever we can. Also, we need to create new components in unexplored parts of the configuration space.

There are two global observations that can be made. First of all, it is not useful (from a complexity point of view) to make connections to nodes that are already in the same connected component. Secondly, it is not useful to connect to nodes that lie too far away. Given these two observations, we can derive the following techniques:

1. Nearest K:
Here we are not going to look at all the neighbors. Instead, we try to connect the sample node $\alpha$ to the nearest K nodes in the graph. The rationale is that nearby nodes lead to short connections that can be checked efficiently. The magic number for K is 15, which works for most situations.

2. Component K (up to K nearest neighbors from each connected component):
Here we try to connect the sample node $\alpha$ to at most K nodes in each connected component. For example, if K=2, the sample node $\alpha$ can connect to at most 2 nodes in each connected component, shown in Figure 5. Component K is a method that trying to bring separate pieces of graphs together.

3. Radius (The distance of neighbor):
Here we change the distance of neighbor to adjust the graph. The tradeoff is already discussed in Section 2.2.

We can even have a merge method of 1 and 3, which means all points within some radius up to K. For example, if 500 nodes are within the radius, the algorithm only picks the top 15. That’s the way restricting the growth.

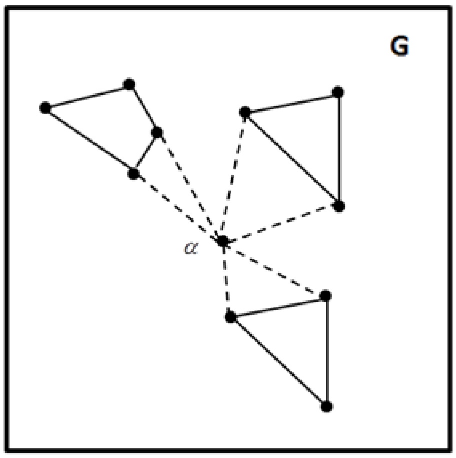

Figure 5 Component K Technique Example (when K = 2)

## 2.4 Lazy-PRM

The basic idea of Lazy-PRM algorithm is to *delay* collision check, until *Inevitable*

Algorithm:
1. Create a dense PRM without ANY collision checking
   When give $q_{init}$ and $q_{goal}$:
   - Find $q_{init} \rightarrow s_1 \rightarrow s_2 \rightarrow q_{goal}$
   - If collision, remove edge

Firstly, assume the C-space is clearly open, and then create a dense PRM without ANY collision checking. When given $q_{init}$ and $q_{goal}$, the algorithm will find the path from the start point through the graph to the goal. This process will be very fast. And then, take the path just found and check collisions. Different from normal PRM, now the program only needs to check the collision just along the path, instead of to check collisions of the whole graph. If any collision is found (the path is broken), then remove the edge and re-plan the path. The algorithm will re-route around the broken edge and repeat the whole process again until it gets a path to the goal.

Lazy-PRM is one way to bridge the gap between high sparsity and doing less collision check. The worst case performance of lazy PRM will be the same as normal PRM. This happens when basically every edge is broken, so that the program will do the collision check for almost all the edges. But the average performance of lazy PRM will be much better than normal PRM.

# 3 Single-Query Algorithm

## 3.1 Random Path Planning / Potential-Field Planner

### 3.1.1 Basic Concept

Before going to the detail of the algorithm, it is necessary to clarify some of the terms. Although different academic papers do not necessarily agree with the terms, the term “Random Path Planning (RPP)” is often used to refer to a family of planning methods. The method discussed in the class sometimes is also referred to as “Potential-Field Planner” [4]. Since the name more closely describes the algorithm, it is used in the following paragraph.

The name “Potential-Field Planner” seems to be a better name because the algorithm actually creates a potential field, which is essentially a function $\boldsymbol{U}$ that maps the object-free configuration space $C_{free}$ to a set of non-negative real number $\boldsymbol{R} \cup \{0\}$, with a single global minimum of zero. The algorithm tries to connect the $q_{init}$ and $q_{goal}$ through two types of motions: *down motion* and *escape motion*. The down motion is named so because the motion travels towards $q_{goal}$, decreasing the value along the potential field represented by $\boldsymbol{U}$; escape motion simply means the motion tries to escape from $C_{obs}$, attempting to avoid getting stuck in the local minima.

### 3.1.2 Algorithm

The following is an example algorithm in pseudo code that implements the Potential-Field Planner. The function Down-Motion keeps looking for adjacent configurations until reaching a configuration $q$ that ends up in the local minimum of the potential field in terms of $U(q)$.

```
Function Down-Motion(configuration: $q_s$): configuration {
    var $q := q_s$
    while ( q is not labeled as a local minimum ) {
        for a trial of $h$ configurations {
            Pick at random a configuration $q'$ adjacent to $q$
            if ($q'$ satisfies $U(q') < U(q)$) { break }
        }
        if (the previous step succeeded in generating $q'$) { $q := q'$ }
        else { Label $q$ as a local minimum }
    }
    return $q$
}
```

The function Escape-Motion moves the current configuration away from the local minimum (i.e. some obstacle) through random walk with a walking distance of *l_desired*.

```
Function Escape-Motion(configuration: $q_l$): configuration {
    Pick at random a length l_desired
    $q$: = $q_l$
    l_current = 0
    while ( l_current < l_desired ) {
        Pick at random a free configuration $q'$ adjacent to $q$
        $l := l + \max(q_i - q_i')$
        $q := q'$
    }
    return q
}
```

The main function Potential-Field-Planner switches between Down-Motion and Escape Motion until reaching the goal configuration

```
Function Potential-Field-Planner(configuration: $q_{init}$, configuration: $q_{goal}$): trajectory {
    $q_l$ := Down-Motion ($q_{init}$)
    while ( $q_l \neq q_{goal}$ ) {
        $q_l' = q_l$
        while ($U(q_l') >= U(q_l)$) {
            if ( the total number of configurations generated so far > threshold ) {
                return None and halt
            }
            $q_s$ = Escape-Motion($q_l$)
            $q_{l'}$ = Down-Motion($q_s$)
        }
    }
    return the trajectory
}
```

### 3.1.3 Discussion

The key of the algorithm is to balance between exploitation and exploration, which are implemented in Down-Motion and Escape-Motion, respectively. The algorithm works well but has some downsides：it requires a lot of parameters tuning and it has poor exploration because of the nature of random walk.

## 3.2 Rapidly Exploring Random Tree

### 3.2.1 Basic Concept

Rapidly Exploring Random Tree (RRT) searches for a path from the initial configuration to the goal configuration by expand a search tree. [5] For each step, the algorithm determines a target configuration and expands the three towards it. The target can either be a random configuration or the goal configuration itself, depends on the probability value defined by the user. During expanding, the algorithm only needs to verify whether each step is collision free but does not need to avoid obstacles.

### 3.2.2 Algorithm

The following is an algorithm implementing the RRT. The function ChooseTarget determines whether the target configuration in the next step of expanding is some random configuration or the goal itself. *p* is a user defined value and for every step, there is a probability of *p* that the target configuration is the goal configuration $q_{goal}$ and a probability of *1-p* that the target configuration is some random configuration.

```
Function ChooseTarget (configuration: $q_{goal}$): configuration {
    Pick a random number uniformly distributed in [0, 1]
    if ( $0 < p < GoalProb$ ) { return $q_{goal}$ }
    else if ( $GoalProb < p < 1$ ) { Return a random configuration }
}
```

The function Nearest determines the node on the tree that is closest to the target configuration $q_{target}$.

```
Function Nearest (tree: $T$, configuration: $q_{target}$): configuration {
    $q_{nearest}$ := $EmptyConfig$
    foreach configuration $q_i$ in T {
        if ( the distance between $q_i$ and $q_{target}$ < the distance between $q_{nearest}$ and
        $q_{target}$ ) { $q_{nearest} := q_i$ }
    }
    return $q_{nearest}$
}
```

The function Extend expands the tree towards a target configuration $q_{target}$ with a unit distance *l*. After expanding, it performs collision checks to determine whether the newly obtained configuration $q_{extended}$ is valid.

```
Function Extend (configuration: $q_{nearest}$, configuration: $q_{target}$): configuration {
    Use some heuristics to extend $q_{nearest}$ towards $q_{target}$ by some distance $l$ to generate $q_{extended}$
    if ( the robot collides with something with $q_{extended}$ ) { return EmptyConfig }
    else { return $q_{extended}$ }
}
```

The function RRTPlan is the main function and repeatedly calls the three functions - ChooseTarget, Nearest and Extend - until reaching the target configuration.

```
Function RRTPlan(configuration: $q_{init}$, configuration: $q_{goal}$): tree {
    $q_{nearest} := q_{init}$
    Initialize a tree $T$
    while ( the distance between the $q_{nearest}$ and $q_{goal}$ < threshold ) {
        $q_{target}$ := ChooseTarget ($q_{goal}$)
        $q_{nearest}$ := Nearest ($T$, $q_{target}$)
        $q_{extended}$ := Extend ($q_{nearest}$, $q_{target}$)
        if ( $q_{extended}$ != EmptyConfig ) { Add $q_{extended}$ into $T$ }
    }
    return $T$
}
```

In addition, Figure 6 serves as a visualization of different stages of tree expansion.

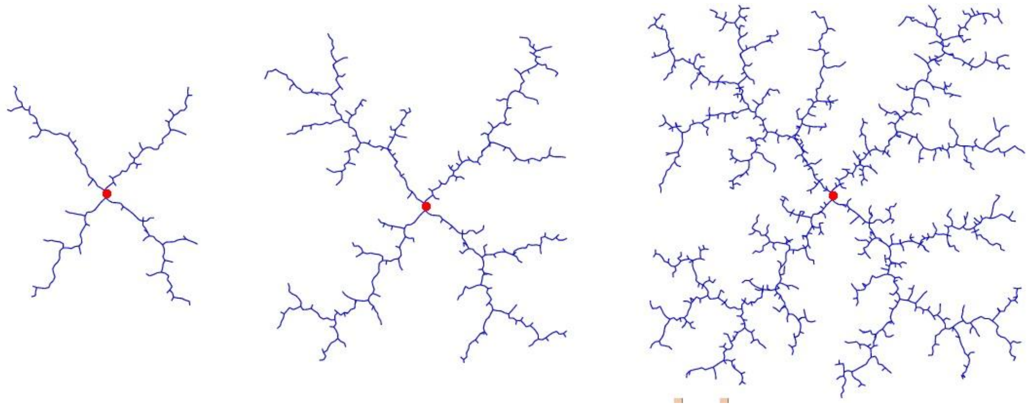

Figure 6 Different Stages of RRT Tree Expansion [6]

### 3.2.3 Possible Improvement

It is worth noting that some improvement can be easily implemented. First of all, if there is only one initial configuration $q_{init}$ and one goal configuration $q_{goal}$, it should be no different between expanding the tree from $q_{init}$ and $q_{goal}$ and expanding in the opposite direction. As a result, one variant of the algorithm is expanding two trees from $q_{init}$ and $q_{goal}$, respectively towards each other, which should speed up the process. However, this method speeds up searching at the cost of loss of generality, because for a region of goals, expanding a tree for each goal configuration is inefficient.

Applying path shortening is another possible improvement. Looking at Figure 6, it can be predicted that the final path consists of a large amount of short lines connected by turns. Applying the following path shortening algorithm may significantly improve the final path.

```
Function ShortenPath (tree: $T$): tree {
    while ( some threshold ) {
        Select two nodes $q_i$ and $q_j$ form $T$ at random
        Connect $q_i$ and $q_j$ with a straight line
        collision := The result of collision checks on the line connecting $q_i$ and $q_j$
        if ( collision == True ) { continue }
        else { Replace the path between $q_i$ and $q_j$ by a straight line }
    }
}
```

### 3.2.4 Discussion

RRT is the first randomized path planning technique particularly designed for handling differential constraints. However, it still has some drawbacks: firstly, the algorithm is sensitive to the metric for evaluating the a configuration; secondly, it searches for nearest neighbors with linear time while some advanced nearest-neighbor searching methods can compute the nearest neighbor in near-logarithmic time; lastly, the algorithm bases itself on random sampling, which may yield relatively poor performance for a simple problem. [7]

## Reference

[1] Kavraki L E, Svestka P, Latombe J C, et al. Probabilistic roadmaps for path planning in high-dimensional configuration spaces[J]. Robotics and Automation, IEEE Transactions on, 1996, 12(4): 566-580.
[2] Geraerts R, Overmars M. A comparative study of probabilistic roadmap planners[J]. Algorithmic Foundations of Robotics V, 2004: 43-58.
[3] Lindemann S R, LaValle S M. Current issues in sampling-based motion planning[J]. Robotics Research, 2005: 36-54.
[4] Barraquand J, Kavraki L, Latombe J-C, Li T-Y, Motwani R, Raghavan P, A Random Sampling Scheme for Path Planning.
obtained at: robotics.stanford.edu/~latombe/papers/ijrr96/random/paper.ps
[5] Bruce J., Veloso M. Real-Time Randomized Path Planning for Robot Navigation
obtained at: http://www.cs.cmu.edu/~mmv/papers/02iros-rrt.pdf
[6] UIUC, 2d RRT Animation.
obtained at: http://msl.cs.uiuc.edu/rrt/gallery_2drrt.html
[7] Lavalle S., Kuffner J. J., Rapidly-Exploring Random Trees: Progress and Prospects. obtained at: http://dora.eeap.cwru.edu/msb/591/LavKuf01.pdf