Sphere Sampling in CSpace for Motion Planning

Yinan Zhang

March 27, 2014

1 Introduction

Traditionally, sampling based motion methods use discrete data to construct paths for motion tasks. The disadvantage of this kind of implementation is obvious:

1. CSpace is a continuous space, discrete samples cannot represent it well;

2. Many samples are actually not necessary and consume a lot of memory, e.t. samples far away from obstacles.

3. The way to get optimal path using samples in the free space requires building a dense roadmap, like PRM*.

Our method is to sample spheres which contain collision-free area in configuration space. Using these hyper-spheres in c-space, we can:

1. describe configuration space in a continuous way;

2. find an optimal path;

3. save memory by reducing unnecessary samples.

2 Sphere Sampling Algorithm

2.1 Something we know

2.1.1 How to interpret the algorithm?

Our algorithm can be viewed as a combination of RRT and visibility-PRM. In each iteration of our algorithm, we get some more spheres center at the edge of existing ones, it rapidly and randomly grows until converge. Unlike RRT, these spheres build not only a tree rooted at the starting sphere, if we build the overlapping relationship of them, each sphere can also be viewed as a node of a roadmap. In a visibility PRM algorithm, if we limit the range a sample can view to a certain distance, or the distance from itself to the nearest obstacle, we can get pretty much the same result, which is why I view these spheres as a visibility roadmap. Inside each sphere, a robot is free to go anywhere without worrying about collision.

2.1.2 Will the algorithm stop?

As is shown in the algorithm, we get boundary points of an existing sphere by limiting the distance between every two points to be equal to δ. Then the algorithm samples new spheres centered at these points. So the distance between two spheres centers is always larger than or equal to δ.

The problem is equivalant to proof there could be finite number of points put in the space such that the distance between every two points is larger than or equal to δ.

Data: Configuration Space
Result: Spheres that cover most of the free space
s0 = sampleSphere(randomly position);
S = {s0};
// Every two boundary points distance equals to δ
boundaryQueue = a queue that contains all boundary points of s0;
while *boundaryQueue is not empty* **do**
    // Candidate point to sample next sphere.
    point = boundaryQueue.pop();
    if *point not in any spheres* **then**
        si = sampleSphere(point);
        if $\delta \leq si.radius$ **then**
            S = S ∩ {si};
            boundaryQueue.put(all boundary points of si);
        end
    end
end

Algorithm 1: Sphere sampling algorithm

Assume we partition the whole configuration space(including free space and obstacle space) into hyper-cubes. The length of every edge of these hyper-cubes is exactly δ. As long as the configuration space has finite volume, the number of hyper-cubes is finite. Assume the number of hyper-cubes is N, the number of hyper-cubes that is totall or partly in free space is less than N. Thus the number of spheres is finite, and less than N.

So the algorithm will always converge.

2.1.3 δ-clearance

[1] proposed an algorithm, named PRM*, to find a path. It requires the actual optimal path to have δ-clearance so the algorithm can find a approximately optimal path. Yet, the algorithm cannot explicitly control the quality of the path by defining δ. In our alogrithm, we can define the actually value of δ and control the coverage of free-space by sampling spheres.

2.1.4 Find Containing Spheres

In the algorithm, we need to know if a candidate point is inside any spheres. Naively, we need to iterate each existing spheres to determine if a point is inside, which requires $O(n)$ time. This can fortunately be done with less time by hashing spheres to uniform grids of the c-space. Each grid contains references to spheres that intersect/contain the grid. By hasing a candidate point in the same way, we can easily know if a point is inside any spheres already.

(Can we use Locally Sensitive Hashing?)

2.1.5 Inaccurate matric

Suppose we have an inccurate matric which gives us the distance to obstacle: $dist[i] = accurate\_dist[i]/c[i]$, where $1 \leq c[i] \leq upper\_bound$. When sampling at a point near the obstacle, $dist[i] = accurate\_dist[i]/c[i] = \delta$ the algorithm will stop sampling at that point. So the farthest distance from spheres boundary to obstacles is $\delta *$ upper_bound.

If the optimal path has $\delta *$ upper_bound clearance. The algorithm can still cover the path and eventually find it.

2.1.6 How to fast determine if there exist a path between two configurations?

One thing I like particularly about this methods is that it can fast determine if there exist a feasible path between two configurations. Usually in PRM* which builds a very dense roadmap, if we want to know whether there exists a path or not, we should run the A* search on the graph, by iterating all nodes in the graph, A* search will finally tell you it failed. It this case, A* search requires exponential time because it has to iterate all nodes in very depth of the graph.

However, if we view the problem as finding a chain of spheres that contains start and goal configurations at the head and end of the chain, it will cost much less time to search the roadmap built by overlapping spheres. Because the number of spheres will be much less than the number of samples in a dense roadmap, and the number of edges is way less than a dense roadmap, which means even in the worst case, an exponential runtime will be much less than that required by A* searching a dense graph.

2.1.7 Does the shape of actually C-Space matter?

I think the answer is clearly a YES. For example, the C-Space for a two-joint robot arm is donut-shaped. However, I represent it in a square-shaped space by connecting the ends of X and Y axis to the heads of them.

Also, if the moving range of two joints are not both $[0, 2\pi]$, say one is $[0, \pi/100]$, the space would look like a really narrow rectangle. However, I still recommend scalling the space to a cube-like shape with equal length of both dimension. Because after sampling, if we scale the space back to what it was, spheres became ellipses, these ellipses covere more space than the same number of spheres.

Another benefit of transforming the space is we probably don't need to care about the matric in original C-Space, I simply use Euclidean distance.

2.1.8 How to deal with unlinked configuration space?

A C-space can be partitioned into several parts by obstacles, each part is not linked to others, and there is not path from one part to another.

The proposed solution to this problem is exactly what visibility PRM will do. In v-PRM, coverage is measured by the number of attemps failed to get a new sample. For example, if we failed n times to get a new sample, the coverage of the visibility graph to the C-space is 1 - 1/n. Here, I think this method can also be used to solved our problem, given an acceptable failure times, say k, we try to randomly get a configuration that is not in any sphere. If we successfully get a new sample with less than k attemps, run the sphere sampling algorithm again, then we can cover the isolated part of C-space. If we cannot get a new sample within k attemps, our spheres have covered 1 - 1/k of the c-space.

2.1.9 What else can these spheres do?

Bialkowski uses similar spheres in his work [2] to help doing efficient collision detection.

For multi-robot motion planning problem, this spheres, generated in a single robot's configuration space, can be reused for other robots. It saves a lot of time because, we don't need to sample in multi-robot configuration space. This is mentioned in [3]

2.2 Demo

Figure 1 is the physics world a 2-joint robot arm is in. Black circles are obstacles. The robot arm cannot move in X or Y axis, but can rotate two joints in range $[0, 2\pi]$

The figure 2 shows the corresponding configuration space of the robot arm. (NOTE: This is actually a torus-shaped space.) Spheres you can see are sampled by our method.
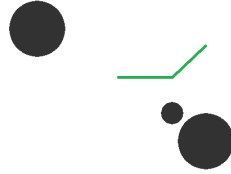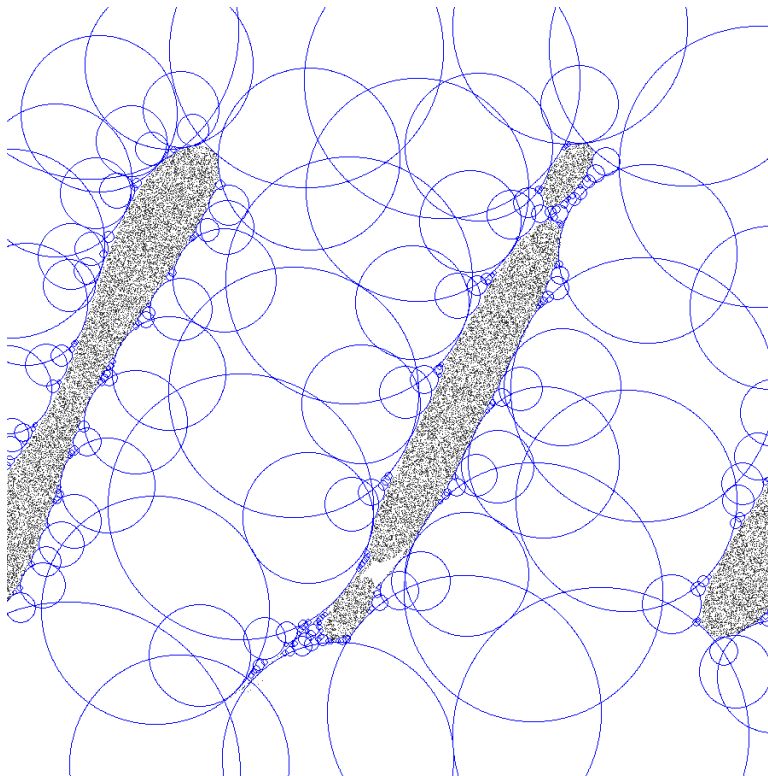
Figure 1: Physics World

Figure 2: C-Space covered by spheres

4

2.3 Some Questions:

2.3.1 How to use less hyper-spheres to cover the space?

As can be seen in Figure 2, in some places there are many annoying small spheres. If these small spheres can be covered by some bigger spheres without breaking the property of containing only free-space, it can save more memory.

2.3.2 How to get the distance to obstacles more accurately?

For now, we shoot several rays from a configuration to all directions in C-Space, then return the shortest distance to obstacles. This is actually not a very good idea, but since everybody else is using it. I didn't hesitate to use it. However, this is not good, actually, it still uses discrete data to represent continuous space. This method is not accurate. If there is a better way to know the distance to nearest obstacles, it would be great.

3 Path search search

We use A* search to find the path.

Data: spheres, start configuration, goal configuration
Result: a path that connects start and end configuratioin.(or fail)
openSet = priority queue containing start;
closedSet = empty set;
while *openSet is not empty* **do**
    current = remove lowest rank point from openSet;
    if *neighbor == goal* **then**
        backtrain(neighbor)
    end
    add current to closedSet;
    neighbors $\leftarrow$ getSuccessors(current);
    add current to closedSet;
    foreach *neighbor in neighbors* **do**
        cost $\leftarrow$ g(current) + movementcost(current, neighbor);
        if *neighbor not in openSet or cost ¡ g(neighbor)* **then**
            parent(neighbor) $\leftarrow$ current;
            g(neighbor) $\leftarrow$ cost;
            heuristic $\leftarrow$ movementcost(neighbor, goal);
            F $\leftarrow$ heuristic + cost;
            openSet.push(neighbor, F);
        end
    end
end

Algorithm 2: Search a path

3.1 Something we know

3.1.1 How to reduce unnessary neighbors

In the method we simpling get all boundary points of a sphere with less than δ distance to each other. Actually, about half of them are not necessary legal neighbors. Some points belong to the sphere their parent is from, these points are useless, because if a path leaves a sphere then enter the same sphere again, it is surely not an optimal path. Giving up these points will save memory as well as run time about 50%.

Data: point, goal
Result: successor points
sphere = get the sphere that contains point;
if *sphere contains goal* **then**
  | return goal.
else
  | return boundary points of sphere, less than δ distance away.
end

<div align="center">Algorithm 3: getSuccessors()</div>

3.2 Something we don't

3.2.1 How to estimate the quality of the path?

I don't know how to make sure the algorithm finds the optimal path. This is something I wrote last time:

"Assuming the path maintains δ clearance, the worst path found by our algorithm is this: the optimal path is in a chain of spheres with radius δ. The point our algorithm chooses to use is δ / 2 away from the actual point the path is going through.

Then the chosen path is 2 times of length of the actual optimal path. "

One thing I can think of is post processing. After getting a path, we can determine if we can connect every other or more points in the path directly, thus avoids redundant distance. (Triangular inequality)

4 Demo

Figure 3 and Figure 4 show two path finding problems solutions in scaled configuration space. Black dots are start and goal configurations. (Remember, the c-space is donut-shaped) Obviously, the path is approximately optiaml.

Corresponding robot movements are shown in Figure 5 and Figure 6:

Video can be find in youtube:

`http://youtu.be/u7BlHCwnKNc` Motion Planning solution 1
`http://youtu.be/8IY8MltdVGI` Motion Planning solution 2

References

[1] Karaman, Sertac, and Emilio Frazzoli, *"Sampling-based algorithms for optimal motion planning"*. The International Journal of Robotics Research 30.7 (2011): 846-894.

[2] Bialkowski, Joshua, et al. *"Efficient collision checking in sampling-based motion planning."*. Algorithmic Foundations of Robotics X. Springer Berlin Heidelberg, 2013. 365-380.

[3] Bialkowski, Joshua, Michael Otte, and Emilio Frazzoli. *"Fast Collision Checking: From Single Robots to Multi-Robot Teams."*. arXiv preprint arXiv:1305.2299 (2013).
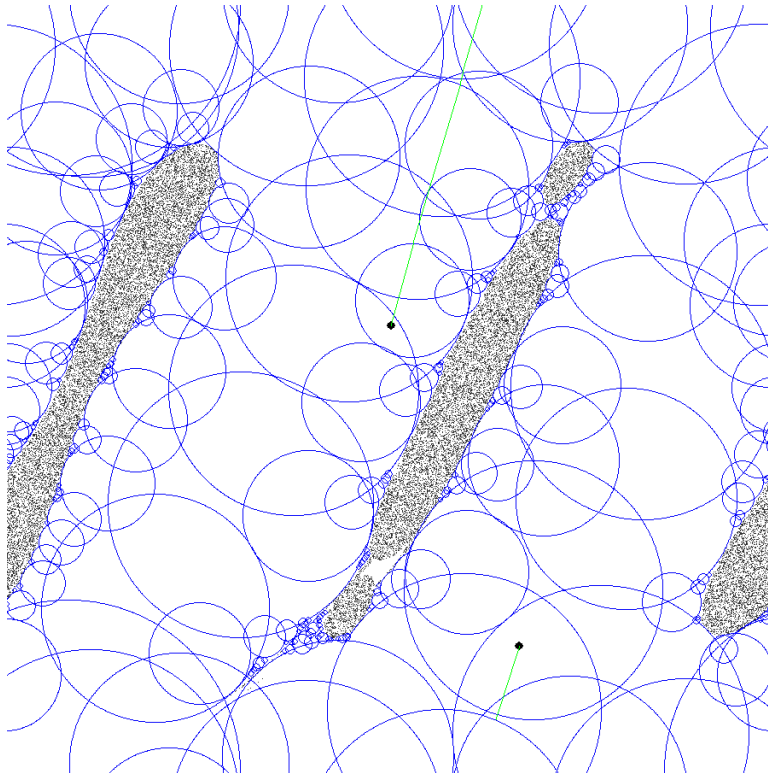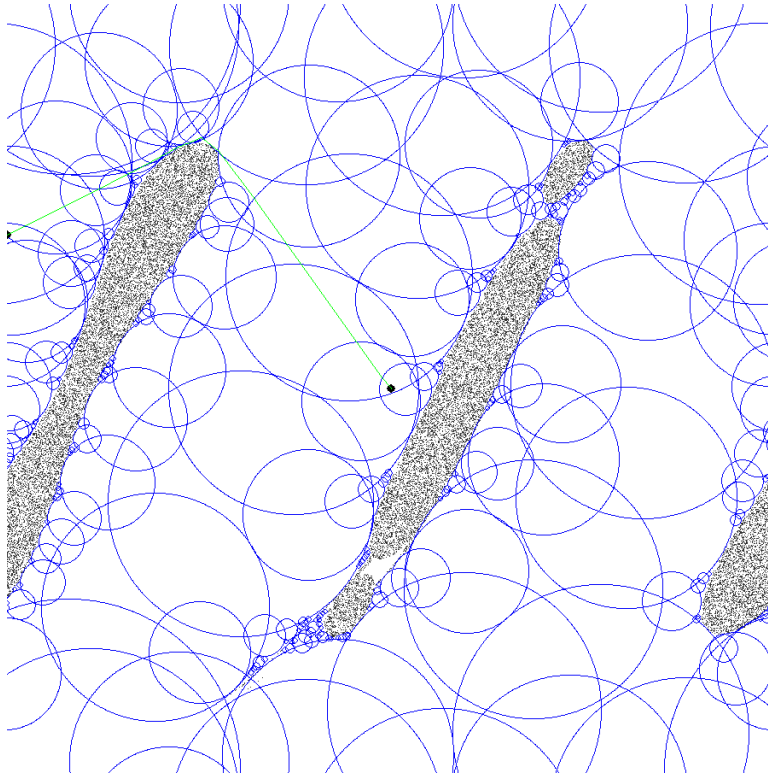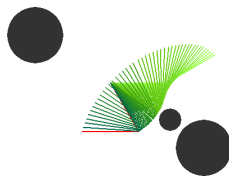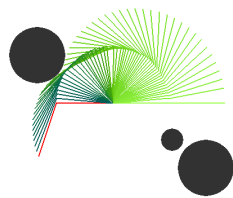
Figure 3: Path finding problem 1

Figure 4: Path finding problems 2

Figure 5: Robot Movement 1

Figure 6: Robot Movement 2