Motion Planning

Junjie Guan < *gjj@cs.dartmouth.edu* >

February 4, 2014

Contents

1 Introduction

Motion planning is a very intersting problem in our realworld. One key issue here is that the metrics beacome infinite in real world, while computer can only handle finite number of states. In this report we introduce some planning methods that is able to build a search tree/graph for real world problem (here the problem is motion planning).

2 Probabilistic Roadmap (PRM)

2.1 Basic Idea

The basic idea of PRM is first sampling the real world and creat a finite configuration space. Then we create a search graph based on the relationship of each configuration (usually a multi-dimensional distance). Then we use traditional path searching algorithm such as A* to find a solution from the start to the goal.

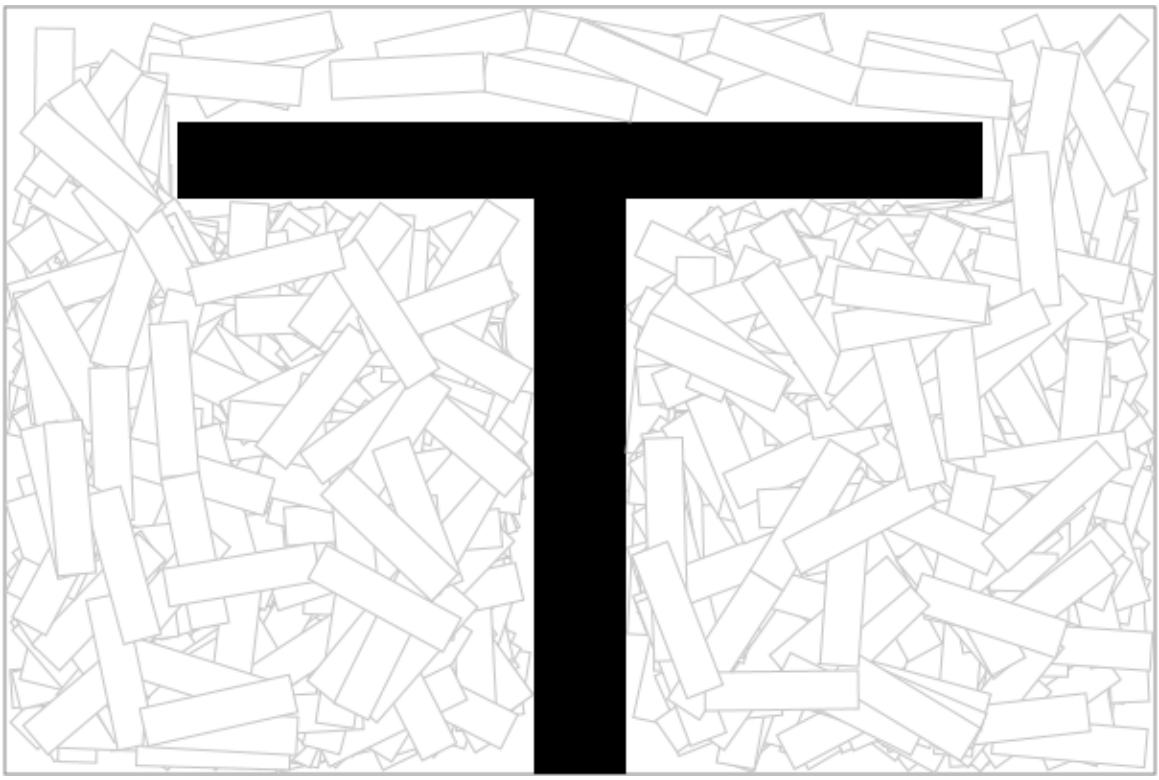

Figure 1: A demostration of A* algorithm

Figure 1 demostrate a sampling space, with all the random configuration of a two-links robot arm (gray rectagnle). The black area is the obstable in the space.

2.2 Discussions

In this homework, in the sampling phase, an intuitive way is to randomly generate the configuration, and check whether it is a legal configuration. If not, dismiss it and keep generate another until a valid one occurs.

An issue we should concern about is, this method usually generate more configuration in a relatively large free space (in motion planning, meaning a space with a few obtacles). This would likely lead to a result that there are not enough configurations in the tricky space (such as a narrow corridor when referring to motion planning problem).

If you look at the figure 1, you will observe that there are significantly less configurations on the top because it is not easy to generat a legal configuration there.

Concerning this issue, we can propose some better sampling method. A simple idea is, every time when we detect a sampling collision, instead of simply dismiss it, we try to move the configuration in a random direction direction until it reaches the edge of the obstacles – becoming a legal configuration. By doing so,

we can make the density of narrow space higher than the normal open space, so that the computer can handle the corner case much better.

2.3 Code implementation

2.3.1 constructor

RoadMapProblem constructor gives you an outline of building the road map.

```
1  public RoadMapProblem(World m, Double[] config1, Double[] config2,
       int density, int K) {
3    // initiate the basic parameters
     k_neighbour = K;
5    map = m;
     startArm = new ArmRobot(config1);
7    goalArm = new ArmRobot(config2);
     startNode = new RoadMapNode(startArm, 0.);
9    // start sampling and generating the roadmap
     getSampling(density);
11   getConnected();
   }
```

2.3.2 getSampling

getSampling is used to create the sample configurations, preparing for roadmap generation.

Line 3: Generate the random configuration. **Line 5:** Check if the newly created configuration is legal. Noted that I modify the the armCollision function, so that the configuration will be confined into the world space. **Line 6:** Add the new configuration to the hash set.

```
public void getSampling(int density) {
2    while (density > 0) {
       Double[] rConfig = getRandCfg(startArm.links, map);
4      ArmRobot toBeAdded = new ArmRobot(rConfig);
       if (!map.armCollision(toBeAdded)) {
6        samplings.add(toBeAdded);
         density--;
8      }
     }
10 }
```

2.3.3 getRandCfg

getRandCfg is used to generate the new random configuration.

Line 3: noted that the length or each arm is fixed.

```
private Double[] getRandCfg(int num, World map) {
2    Random rd = new Random();
     Double[] cfg = new Double[2 * num + 2];
4    // randomize the start position
     cfg[0] = rd.nextDouble() * map.getW();
6    cfg[1] = rd.nextDouble() * map.getH();
     // System.out.println(cfg[0]/map.getW() + "," + cfg[1]/map.getH());
```

```
  for (int i = 1; i <= num; i++) {
    // the length of each arm remains the same
    cfg[2 * i] = startArm.config[2 * i];
    // randomize the angle
    cfg[2 * i + 1] = rd.nextDouble() * Math.PI * 2;
  }
  return cfg;
}
```

2.3.4 getConnected

getConnected is used to connect the sampling. Basic idea is iterating through the samples and connect to k nearest vertice in the roadmap. The raodmap is implemented in adjacent linked list style.

```
public void getConnected() {
  // initiate the connecting with start arm
  ArmLocalPlanner ap = new ArmLocalPlanner();
  PriorityQueue<AdjacentCfg> tmpq;
  for (ArmRobot ar : samplings)
    roadmap.put(ar, new PriorityQueue<AdjacentCfg>());

  // add the start and goal to the roadmap
  connected.add(startArm);
  connected.add(goalArm);
  for (ArmRobot ar : samplings) {
    for (ArmRobot arOther : connected) {
      if (ar != arOther
          && !map.armCollisionPath(ar, ar.config, arOther.config)) {
        Double dis = ap.moveInParallel(ar.config, arOther.config);
        roadmap.get(ar).add(new AdjacentCfg(arOther, dis));
        // make sure the configuration only points to k nearest neighbour
        if (roadmap.get(ar).size() > k_neighbour)
          roadmap.get(ar).poll();
        // make sure that it an undirected graph
        roadmap.get(arOther).add(new AdjacentCfg(ar, dis));
        if(roadmap.get(arOther).size() > k_neighbour)
          roadmap.get(arOther).poll();
      }
    }
    // keep track of the connected configurations
    connected.add(ar);
  }
}
```

2.3.5 boundary detect

boundary detect makes sure that the configuration do not escape the world area.

```
...
for (int j = 1; j <= p.getLinks(); j++) {
  link_i = p.getLinkBox(j);
```

```
  for(int i = 0; i < link_i.length; i++){
  if(link_i[i][0] < 0 || link_i[i][0] > window_width)
    return true;
  if(link_i[i][1] < 0 || link_i[i][1] > window_height)
    return true;
  }
}
...
```

2.4 Output demonstration

Figure 2 demostrates the start and and end configuration of my testing. Figure 3-5 presents 3 examples of the testing case, with increasing sampling difficulty. The number of sample configuration is 500, and k is 15.

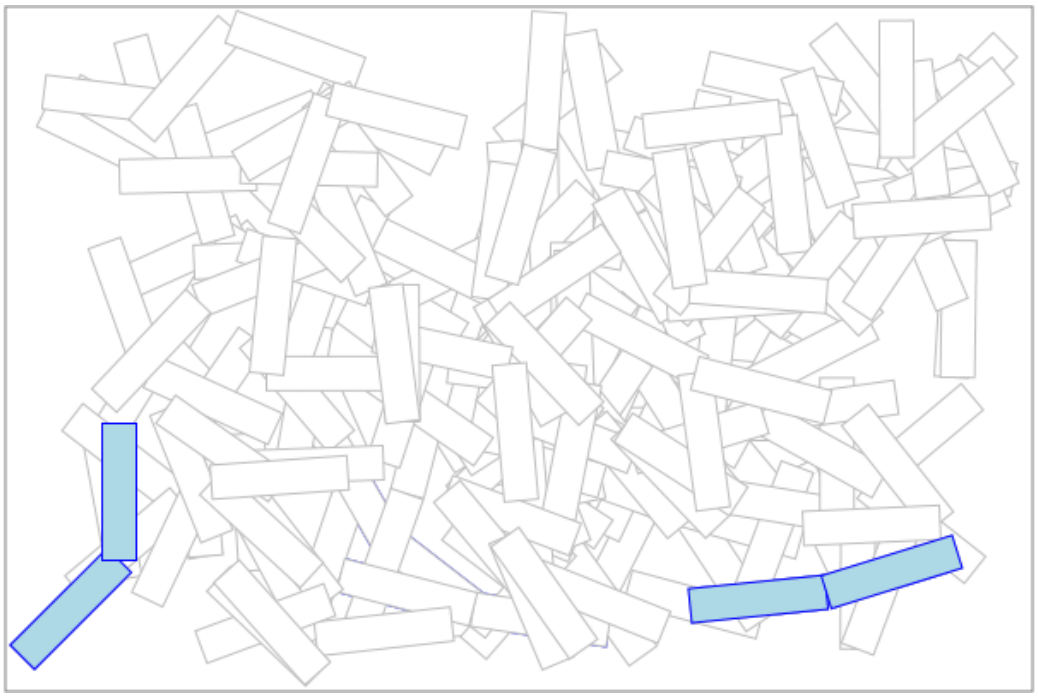

Figure 2: start and end of testing

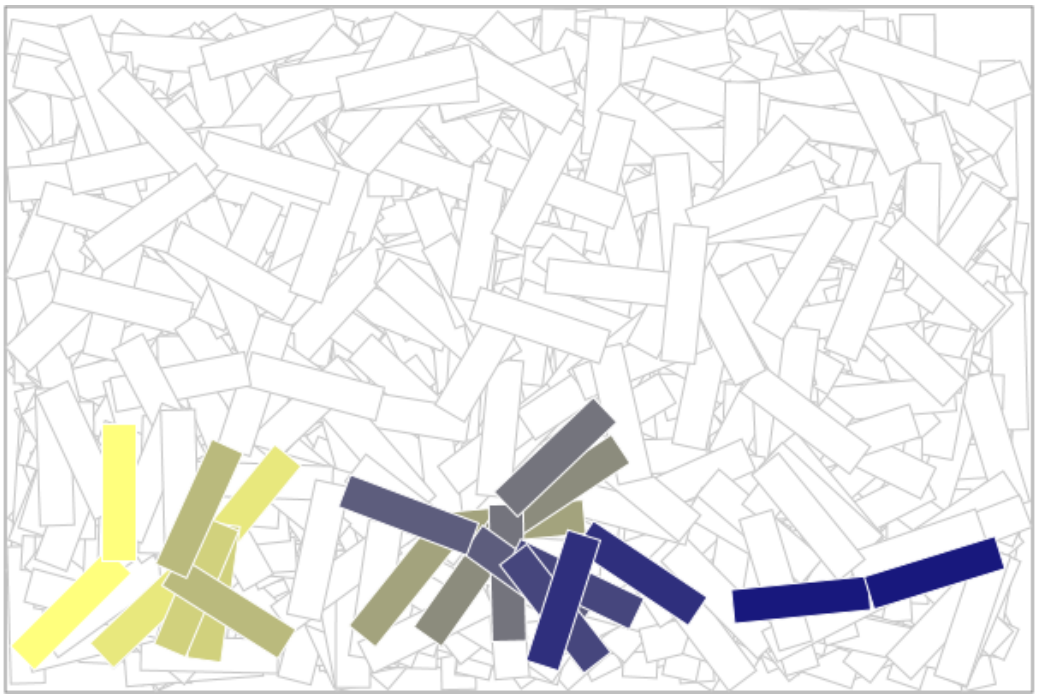

Figure 3: start and end of testing

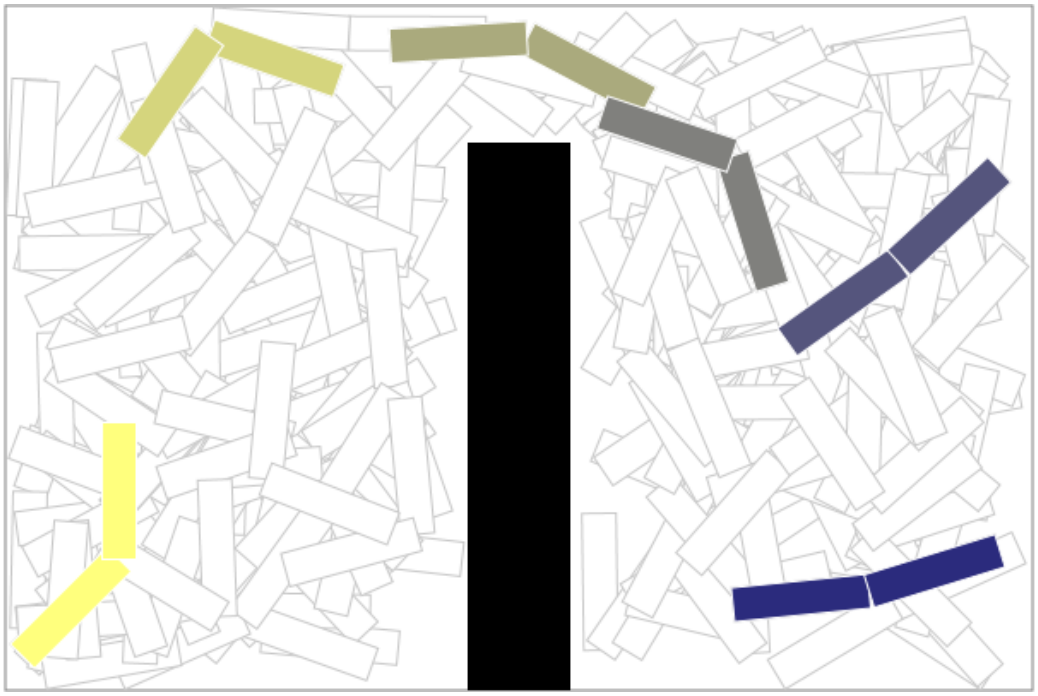

Figure 4: start and end of testing

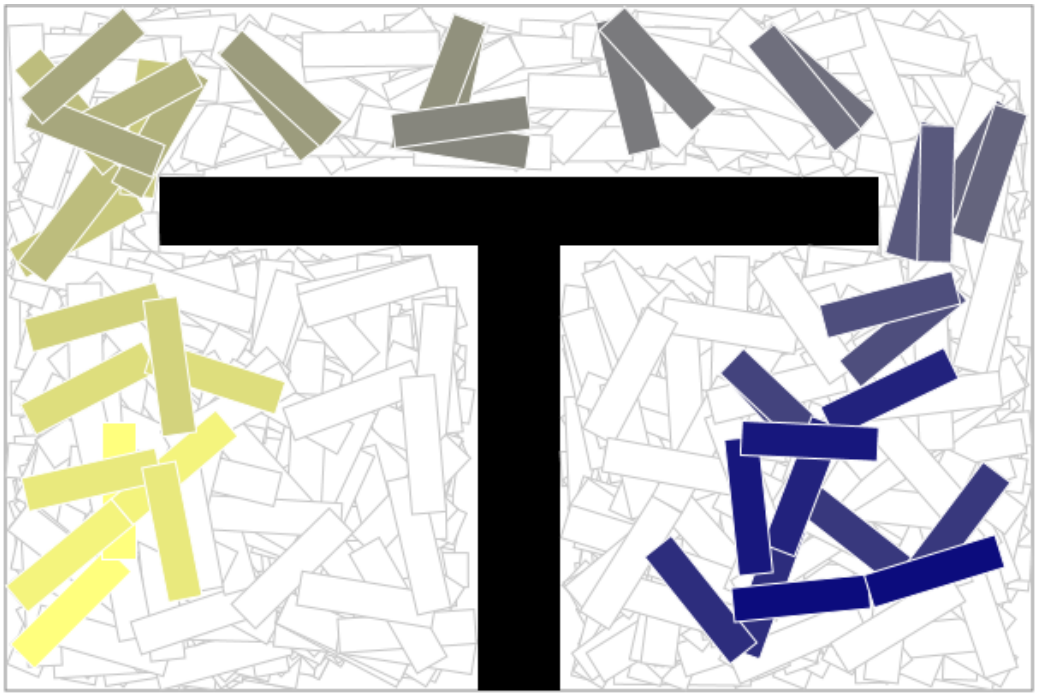

Figure 5: start and end of testing

3 Rapidly Exploring Random Tree