Motion Planning

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

February 5, 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.

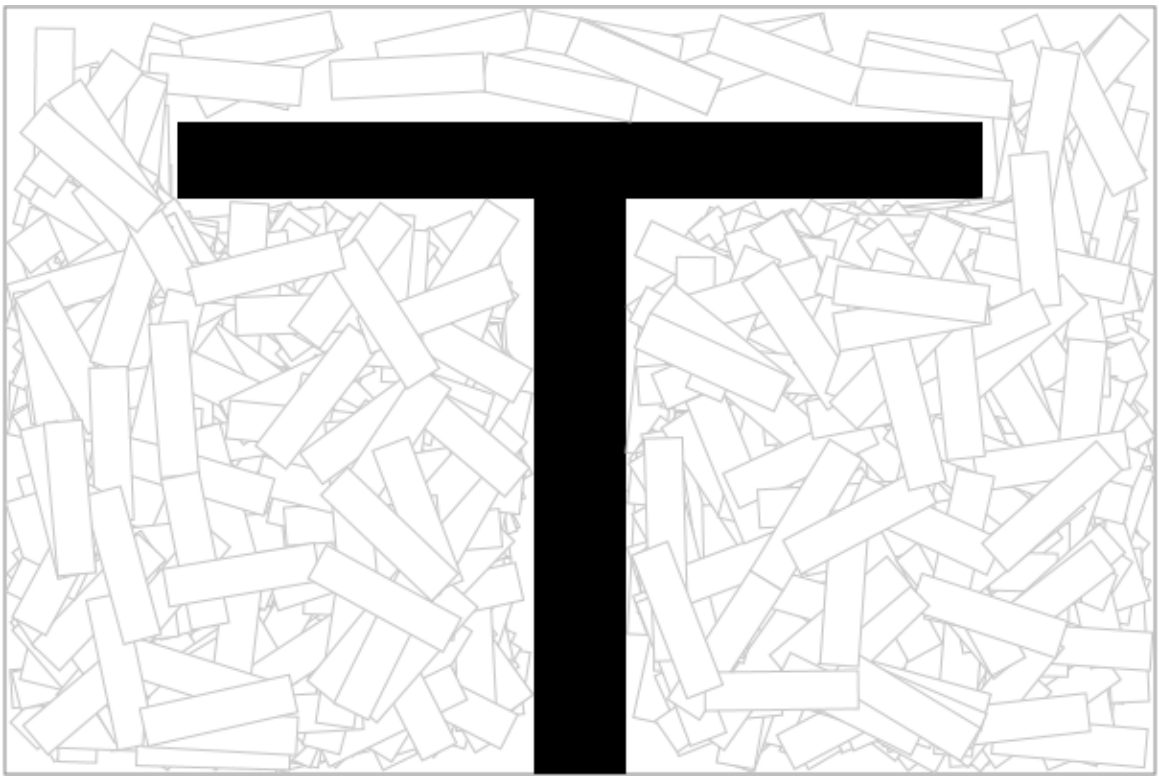

Figure 1: A demostration of A* algorithm

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.

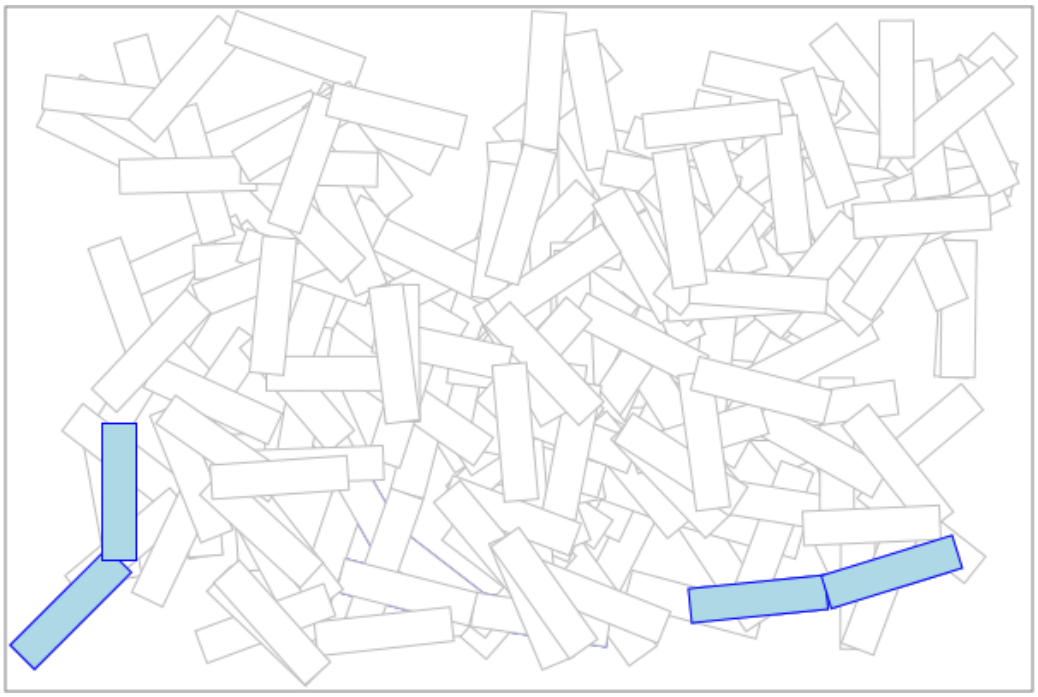

Figure 2: start and end of testing

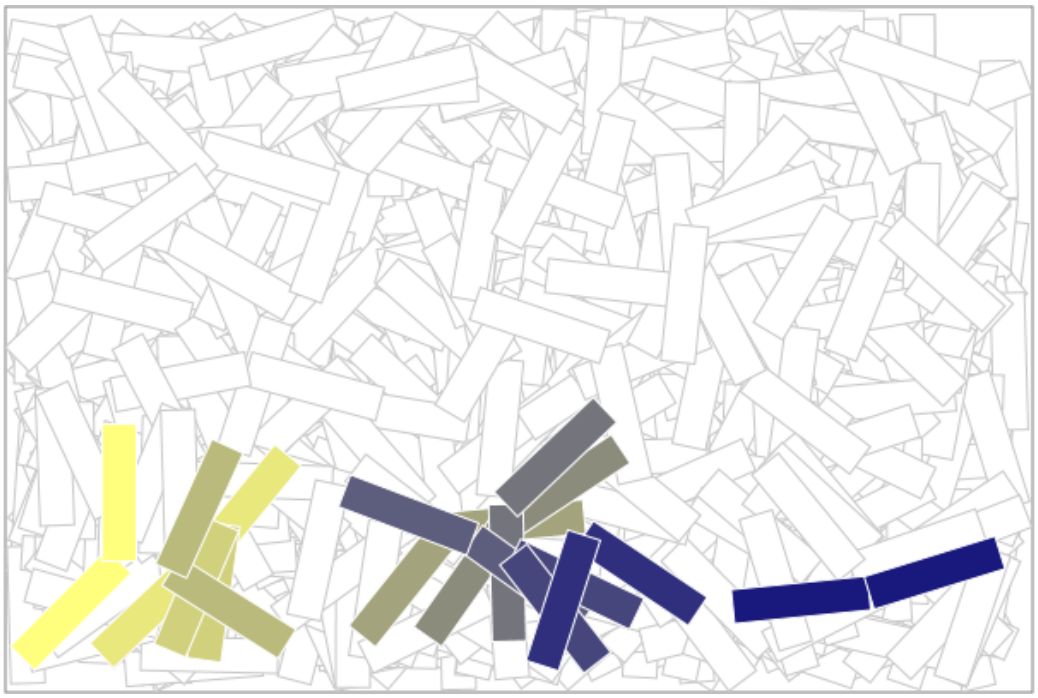

Figure 3: test in empty space

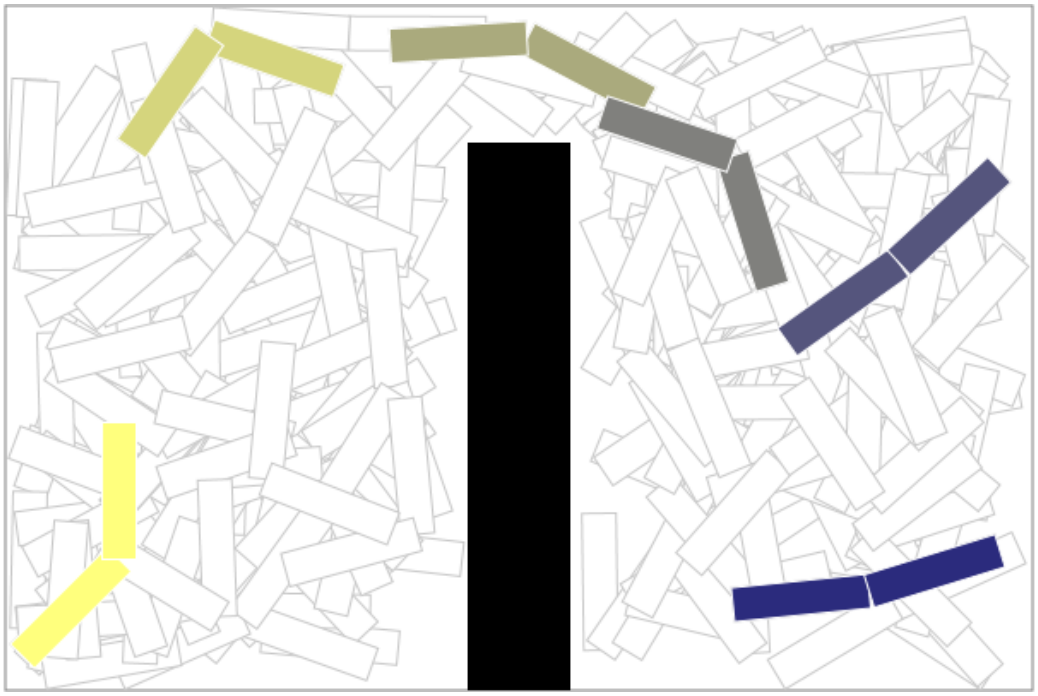

Figure 4: test with an obstacle.

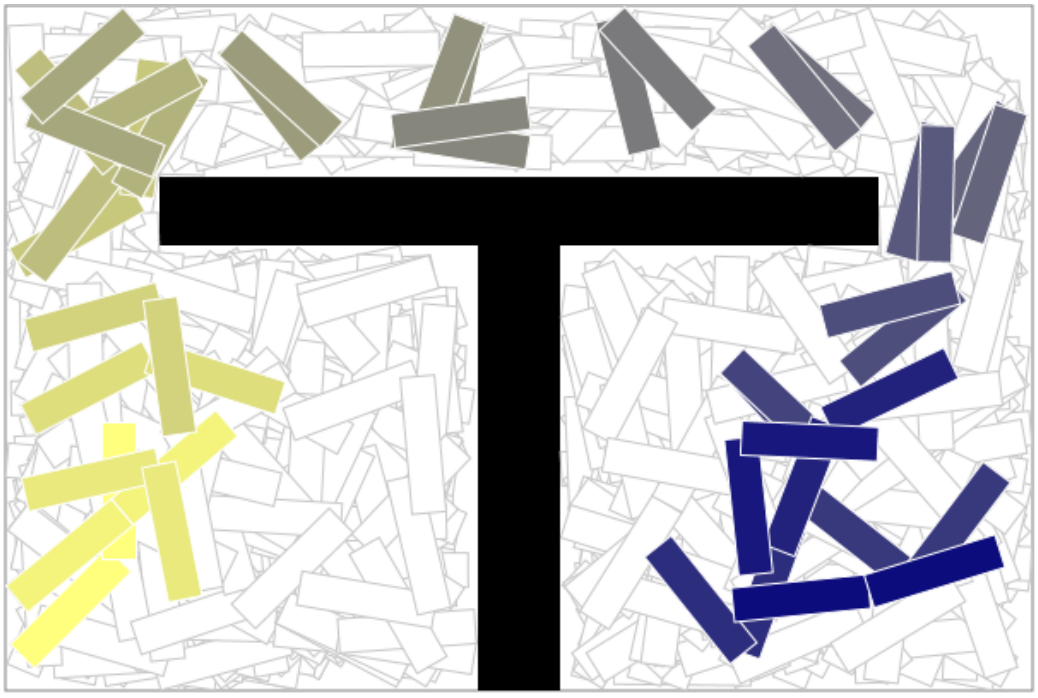

Figure 5: test with corridor. The movement becomes more tricky and complex

3 Rapidly Exploring Random Tree

3.1 Basic Idea

The basic idea of RRT is to grow a tree from the start, to fill up the infinite or over complex space. The most simple and intuitive way is to randomly generate legal configuration, and grow the tree towards the configuration. By the way, the heuristic here is simply the distance from current configuration to goal configuration.

3.2 Code implementation

3.2.1 growTree2Goal

growTree2Goal keeps interatively expanding the tree until one of the node close to the goal. (I think the code is already quite self-explanatory, no further introduction is needed)

```
public void growTree2Goal(int num4grow) {
  while (num4grow > 0) {
    // generate new random car
    CarRobot newRandCar = new CarRobot(getRandCfg(map));
    // if the car is valid
    if (!map.carCollision(newRandCar)) {
      CarRobot nearest = findNearestInTree(newRandCar);
      CarRobot newAdded = expandTree(newRandCar, nearest);
      if(!connected.contains(newAdded)) {
        addNewNode2Tree(newAdded, nearest);
        // terminate the iteration if reaching the goal
        if (newAdded.getDistance(goalCar) < 20) {
          addNewNode2Tree(goalCar, newAdded);
          break;
        }
        num4grow--;
      }
    }
  }
}
```

findNearestInTree finds the vertex that is nearest to the new randomly generated node.

```
private CarRobot findNearestInTree(CarRobot newRandCar) {
  Double minDis = Double.MAX_VALUE;
  CarRobot nearest = null;
  for (CarRobot cr : connected) {
    double dis = newRandCar.getDistance(cr);
    if (minDis > dis) {
      minDis = dis;
      nearest = cr;
    }
  }
  return nearest;
}
```

expandTree trys to expand to 6 different directions, and return the expansion that is nearest to the randomly generated node.

```
private CarRobot expandTree(CarRobot newRandCar, CarRobot nearest) {
  // initiate the auxiliary object
  Double minDis = Double.MAX_VALUE;
  SteeredCar sc = new SteeredCar();
  CarRobot newAdded = null, newCarRobot = new CarRobot();
  // try to expand to 6 different directions
  for (int i = 0; i <= 5; i++) {
    newCarRobot = new CarRobot(sc.move(nearest.getCarState(), i, 1.));
    // System.out.println("newCarRobot: " + newCarRobot);
    if (!map.carCollision(newCarRobot)) {
      double dis = newCarRobot.getDistance(newRandCar);
      if (minDis > dis) {
        minDis = dis;
        newAdded = newCarRobot;
      }
    }
  }
  return newAdded;
}
```

addNewNode2Tree trys to expand to 6 different directions, and return the expansion that is nearest to the randomly generated node.

```
private void addNewNode2Tree(CarRobot newNode, CarRobot parentNode) {
  connected.add(newNode);
  RRTree.get(parentNode).add(new AdjacentCfg(newNode, 1));
  RRTree.put(newNode, new HashSet<AdjacentCfg>());
}
```

3.2.2 getSuccessors

getSuccessors is pretty simple. Simply return the children nodes in the exploring tree.

```
public ArrayList<SearchNode> getSuccessors() {
  ArrayList<SearchNode> successors = new ArrayList<SearchNode>();
  for (AdjacentCfg adj : RRTree.get(car)) {
    successors.add(new RRTnode(adj.ar, adj.dis + cost));
  }
  return successors;
}
```

3.2.3 getDistance

getDistance computes the distance between two configuration. I use a function similar to Euler distance while also considering the angle, using a parameter to tune the effect of angle. What's more, we need to be very careful when implementing the difference of two angle., as line 2-3 show.

```
public double getDistance(CarRobot cr) {
  double angleDif = Math.abs(s.getTheta() - cr.getCarState().getTheta());
  angleDif = angleDif >= Math.PI ? 2 * Math.PI - angleDif : angleDif;
  return Math.pow(
```

```
        Math.pow(s.getX() - cr.getCarState().getX(), 2)
            + Math.pow(s.getY() - cr.getCarState().getY(), 2) + 100
            * angleDif, 0.5);
}
```

3.3 Output demonstration

Figure 6-8 presents 3 examples of the testing case, with increasing sampling difficulty. From the last two case we can observe that, the tree on left part is much more complicate. This is because the left part has keep randomly generating the tree when one of the branch trying to explore through the corridor. I think using a bidirectional spaning can cut the expense of exploring corridor to half.

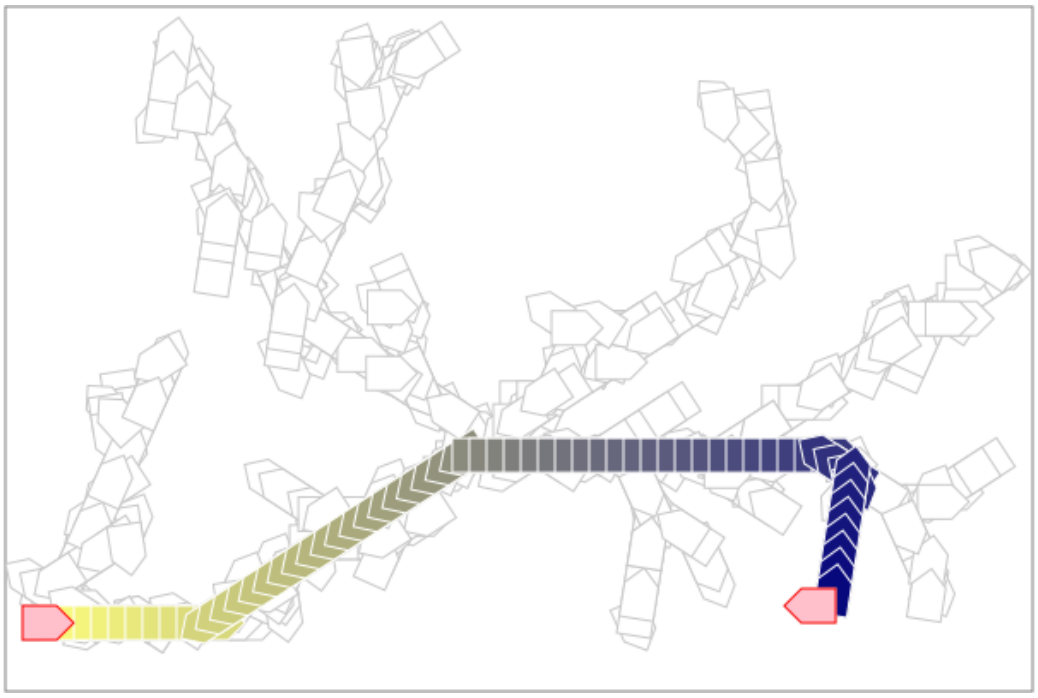

Figure 6: test in empty space

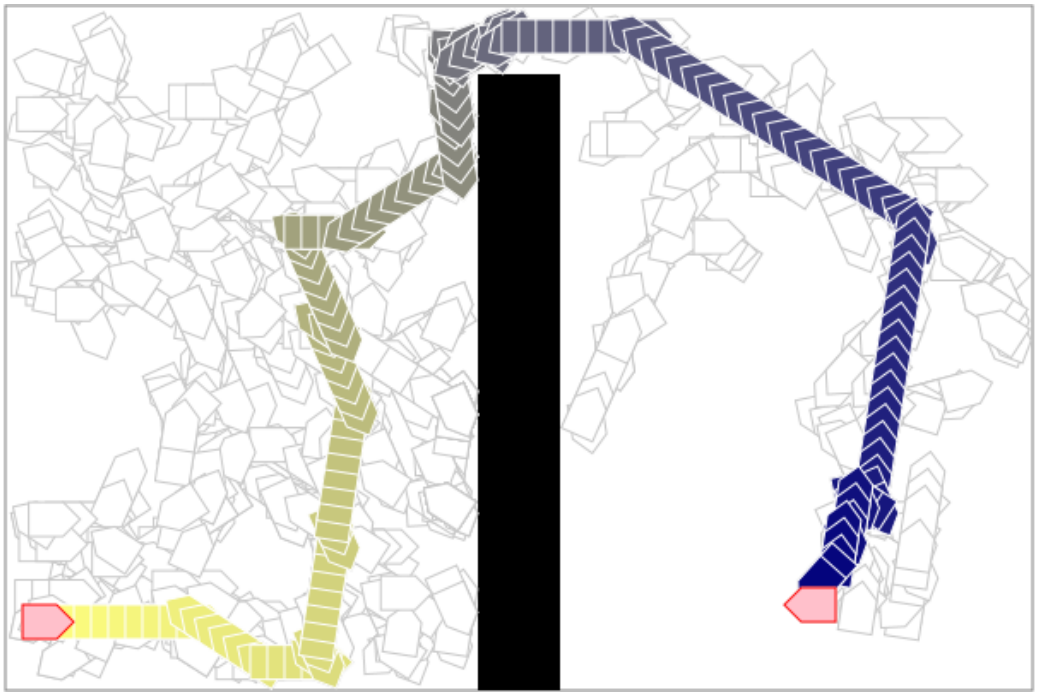

Figure 7: test with an obstacle.

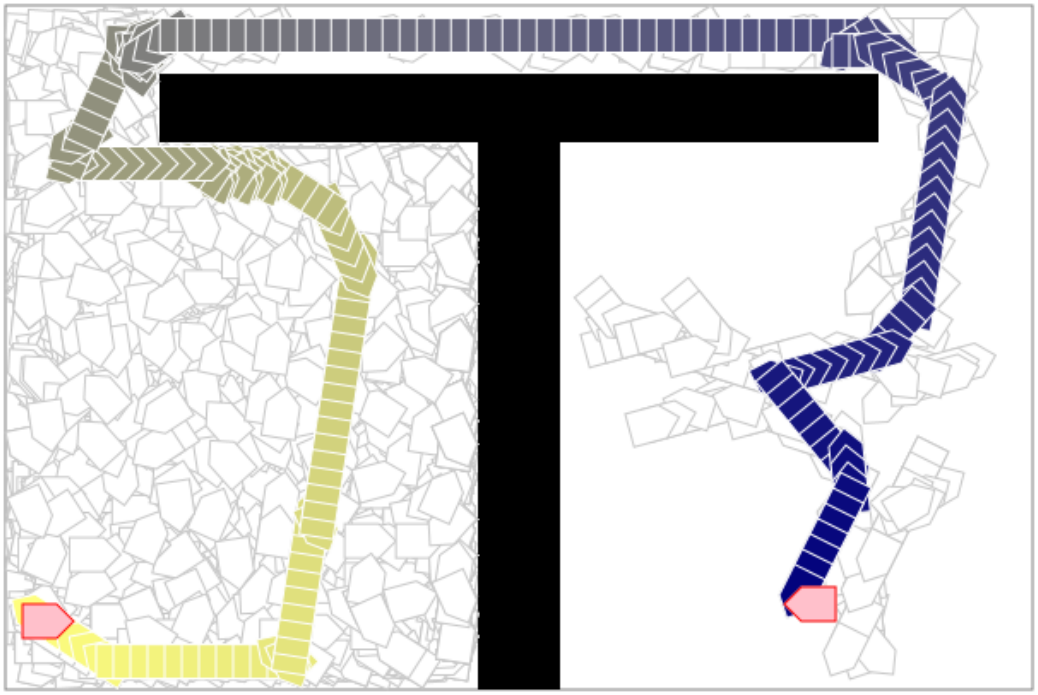

Figure 8: test with corridor. The movement becomes more tricky and complex. The tree size recorded for this time is 11352, and the number of other testing should be fluctuating around it. I specify the tree size for newAddedrison in my latter work, which will be mentioned soon

4 Extension: bidirectional search

4.1 Basic Idea

The essense of of bidiretional search is constructing the searching tree from both start and goal vertex, so that the total tree size can be reduced at least 50%. Also it is benefit to some situation when there is a bug trap preventing searching from source to goal or goal to source. In short it is very powerful.

4.2 Code implementation

Implementing the code is a little tricky. For example we should be careful about how to connect this two tree together when they meet. Also, we should care about the tree structure of these two trees.

In my case I implement tree using singly-linked way. So when building RRT I build a tree with each parent points to its chilren nodes. But in this double tree problem, we might want to implement the goal tree in a reverse way. Of course, you can use doubly-linked style tree and free of considering those issue, but it takes you approximately twice as much as memory to store the tree.

4.2.1 biGrowTree2Goal

biGrowTree2Goal is similar to **growTree2Goal**. The difference is after adding a new node to current expanding tree, we should guide the other tree to expand towards the new node we just added (line 17). And after trying to add new node to both of the tree, we should also keep balance of this two spanning tree, and try to use new random node to expand a smaller tree (line 22).

```
public void biGrowTree2Goal() {
  boolean tree = TreeA;
  while (num4grow > 0) {
    // generate new random car
    CarRobot newRandCar = new CarRobot(getRandCfg(map));
    // initiate current tree set
    HashSet<CarRobot> connected = tree ? connectedA : connectedB,
                theOther = tree ? connectedB : connectedA;
    // if the car is valid
    if (!map.carCollision(newRandCar)) {
      CarRobot nearest = findNearestInTree(newRandCar, tree);
      CarRobot newAdded = expandTree(newRandCar, nearest);
      if (!isContains(connected, newAdded)) {
        addNewNode2Tree(newAdded, nearest, tree);
        num4grow--;
        // terminate the iteration if two tree connected
        if (growTheOther(!tree, newAdded)) break;
      }
    }
    // swap if current tree size exceeds the other
    //System.out.println(tree);
    if(connected.size() > theOther.size()) tree = !tree;
  }
}
```

growTheOther is a method for tree to expand based on the newly added noded in the other tree. And it is also the place to determine whether these two tree are ready to connected to each other (line 8-20).

```
boolean growTheOther(boolean tree, CarRobot target) {
```

```
  HashSet<CarRobot> connected = tree ? connectedA : connectedB;
  CarRobot nearest = findNearestInTree(target, tree);
  CarRobot newAdded = expandTree(target, nearest);
  if (!isContains(connected, newAdded)) {
    addNewNode2Tree(newAdded, nearest, tree);
    // terminate the iteration if reaching the goal
    if (newAdded.getDistance(target) < 20) {
      if(tree) {
        bridgeAside = newAdded;
        bridgeBside = target;
      }
      else {
        bridgeAside = target;
        bridgeBside = newAdded;
        RRTreeB.put(newAdded, new HashSet<AdjacentCfg>());
        RRTreeB.get(newAdded).add(new AdjacentCfg(nearest, 1));
      }
      return true;
    }
    num4grow--;
  }
  return false;
}
```

addNewNode2Tree is a little tricky since two tree is contructed in a opposite way. See the difference between line 7-8 and line 11-12.

```
private void addNewNode2Tree(CarRobot newNode, CarRobot parentNode,
   boolean tree) {
  HashSet<CarRobot> connected = tree ? connectedA : connectedB;
  HashMap<CarRobot, HashSet<AdjacentCfg>> RRTree = tree ? RRTreeA : RRTreeB;
  connected.add(newNode);
  if(tree) {
    // in treeA, parent points to its children, 1 to n
    RRTree.get(parentNode).add(new AdjacentCfg(newNode, 1));
    RRTree.put(newNode, new HashSet<AdjacentCfg>());
  }else {
    // in treeB, the child points to their parent, 1 to 1
    RRTree.put(newNode, new HashSet<AdjacentCfg>());
    RRTree.get(newNode).add(new AdjacentCfg(parentNode, 1));
  }
}
```

The rest of the methods are pretty much similar, with additional tree controling using a parameter.

4.2.2 getSuccessors

getSuccessors is a little more complicated than before, because it acts differently in Tree A and Tree B.

```
public ArrayList<SearchNode> getSuccessors() {
  HashMap<CarRobot, HashSet<AdjacentCfg>> RRTree = tree ? RRTreeA : RRTreeB;
  ArrayList<SearchNode> successors = new ArrayList<SearchNode>();
  // no brdige, no way
```

```
5     if(bridgeAside == null) return successors;
      for (AdjacentCfg adj : RRTree.get(car)) {
7       if(tree && bridgeAside.getDistance(adj.ar) < 20) {
          successors.clear();
9         successors.add(new BiRRTnode(bridgeBside, cost, !tree));
          return successors;
11      }else {
          successors.add(new BiRRTnode(adj.ar, adj.dis + cost, tree));
13      }
      }
15    return successors;
  }
```

4.2.3 heuristic

heuristic is interesting here. Since the moment you enter Tree B, there will be only one way that leads you to the goal, which is the root of tree B. In a way you already find the goal, and there is no point to keep searching on any other node.

So after entering tree B, I simply set the heuristic to $-cost$, so that I can make sure the prioty is 0, and it always will be the first in the prioty queue of A* searching.

```
  @Override
2 public double heuristic() {
    // if in the treeB, we should simply go straight to the goal
4   // so I set priority as high as possible
    if(tree)
6     return car.getDistance(goalCar);
    else
8     return -cost;
  }
```

4.3 Output demonstration

Figure 6-8 presents 3 examples of the testing case, with increasing sampling difficulty. From the last two case we can observe that, the tree on left part is much more complicate. This is because the left part has keep randomly generating the tree when one of the branch trying to explore through the corridor. I think using a bidirectional spaning can cut the expense of exploring corridor to half.

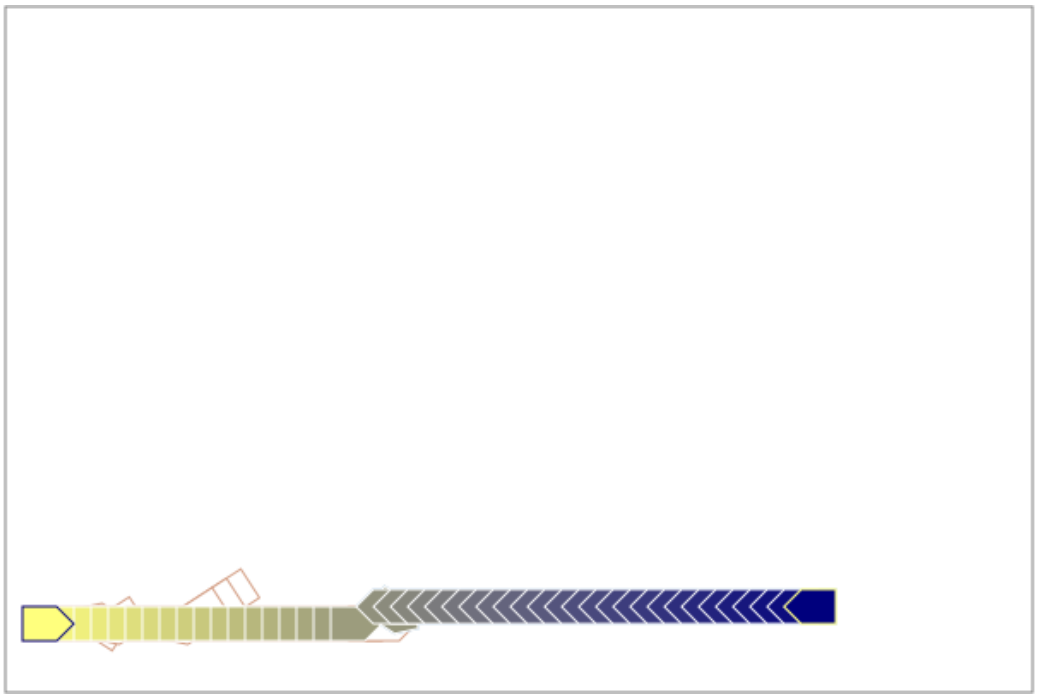

Figure 9: test in empty space

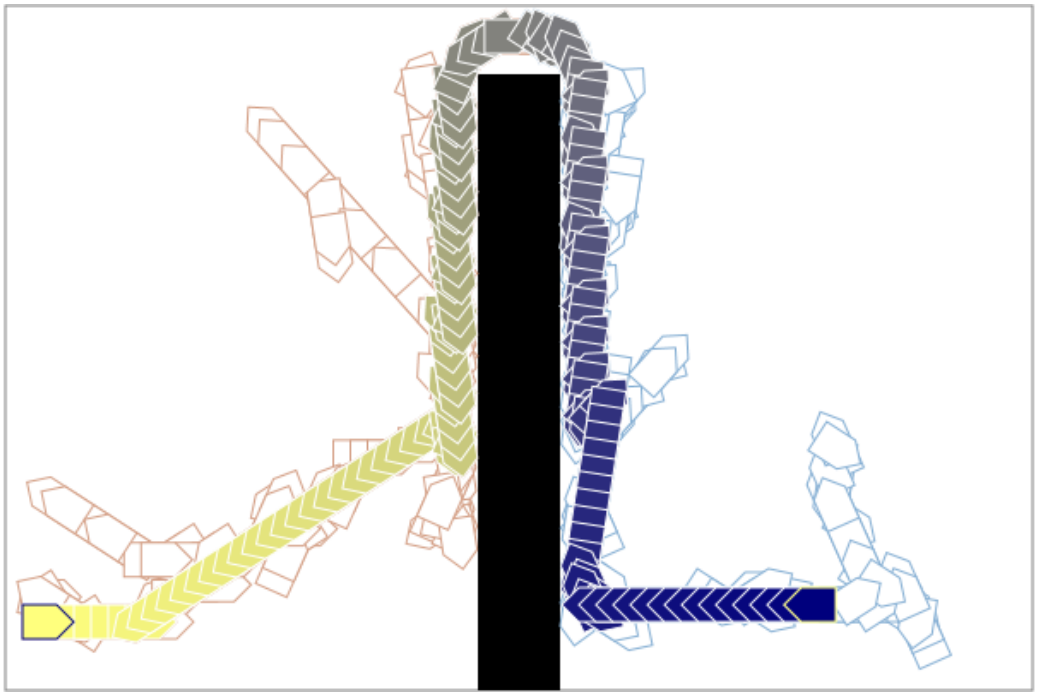

Figure 10: test with an obstacle.

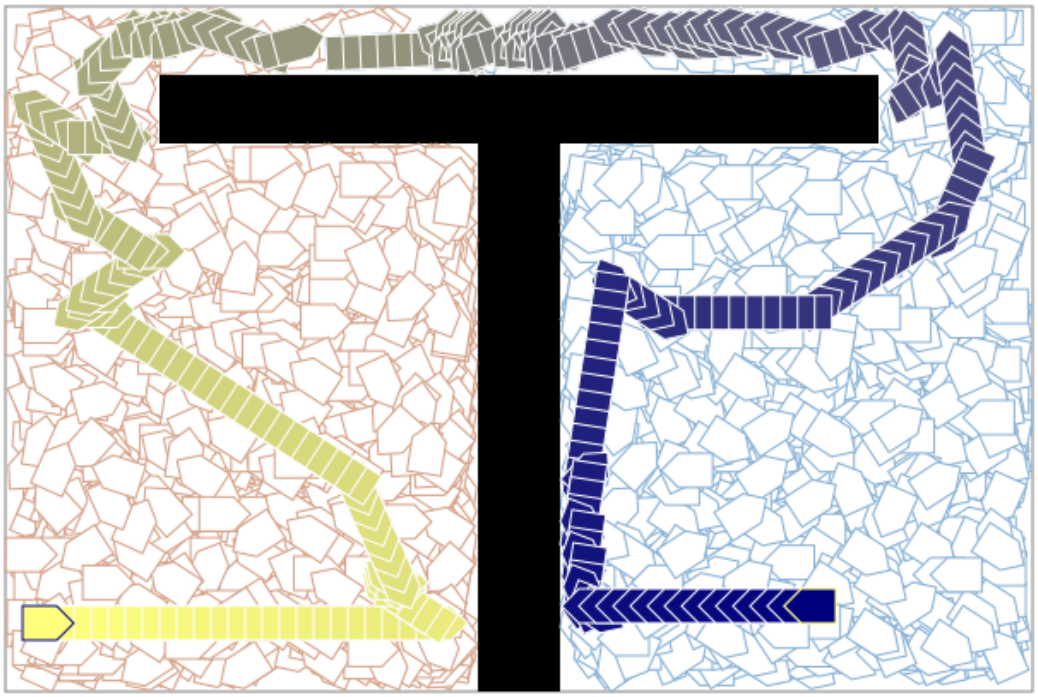

Figure 11: test with corridor. Though the tree looks crowded here. But actually the total size of these two tree is 2370×2. Comparing to figure 8 which is 11352, it uses only around half of the nodes. What's more, the searching is even more efficient, since A* stop exploring all the other nodes as soon as it reach the conjunction node of these two tree.

5 Previous work

RRT-connect [1] is a very interesting research work. Regarding to the issue of building RRT in high-dimensional configuration space, they propose a way to build the RRT from source and goal simultaneously. Some advantages including free of tunning parameters, flexible in solving many difficult problem in realworld.

RRT-connect [2] is also very interesting. In my course I use simple uniform distribution to generate the random node. In this paper, the author proposes to use Gaussian sampling strategy. As I mentioned in my disccussion of sampling method, uniform distirbution totally neglect the high density needs of configuration around the obstacles. While using Gaussian distribution, we can actually focus more around the obstacles instead of the free space. The experiment results shows the efficiency of this algorithm.

[1] Kuffner Jr, James J., and Steven M. LaValle. "RRT-connect: An efficient approach to single-query path planning." Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on. Vol. 2. IEEE, 2000.

[2] Boor, Valrie, Mark H. Overmars, and A. Frank van der Stappen. "The gaussian sampling strategy for probabilistic roadmap planners." Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on. Vol. 2. IEEE, 1999.