Final Project: Maze Solver

Seyed Mahdi Basiri Azad (Erfan)

March 9, 2016

1 Overview

In this project I wanted to explore the RRT (Rapidly Exploring Random Trees), from the Motion Planing project, in more detail. Especially I wanted to implement a **bidirectional RRT** and experiment with **different heuristics** to compare the performance of the algorithm.

2 Heuristics

The heuristic that I considered was introducing a **goal bias** variable such that with a certain probability it would choose the goal configuration as the next "random" configuration. by setting that variable to 0 it would turn off this feature. Increasing the goal bias probability would give this RRT a greedy property while growing the graph. **Figure 1** shows pictures of goal biased probability being 0.0 (no goal bias) and 1.0 (greedy). Although increasing the goal bias creates a more straight forward path from start to the goal by increasing the greediness of the algorithm its usage depends on the application. The majority of time for the RRT algorithm is used to grow the graph so it would be wise to grow the graph and store it so that

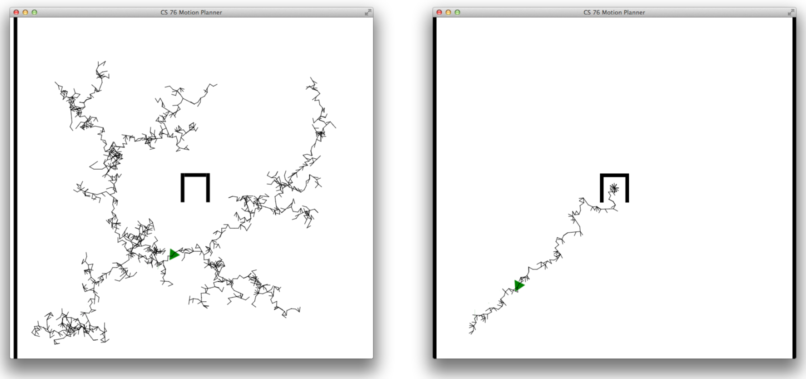
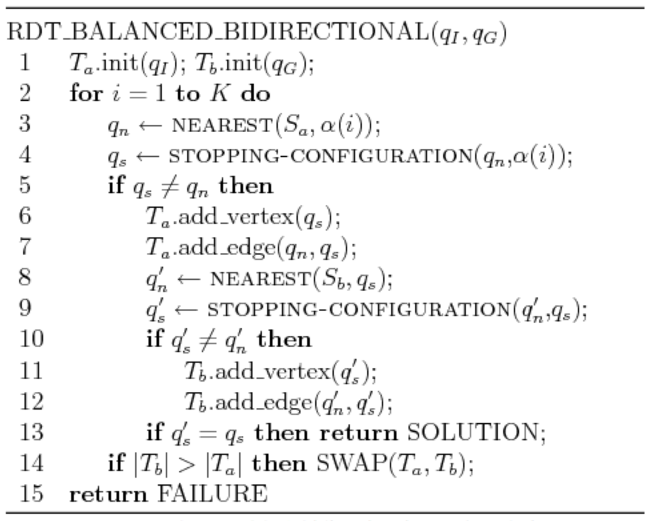


Figure 1: Goal Bias of 0.0 (Left) and 1.0 (Right)

more searches can be done using the information that the already grown graph provides. Increasing the goal bias will reduce the deviation on the areas that are explored and will concentrate more on the straight path between start and goal. In other words increasing the goal bias too much will over-fit the graph to a particular search and hence will reduce the performance of the search on new searches.

3 Balanced-Bidirectional-Search

I decided to expand the motion planing assignment by implementing the Balanced-Bidirectional-Search for RRT. I used the psudocode from Steven LaValle's Planing Algorithms (`http://planning.cs.uiuc.edu/node237.html`) for writing the search.



```
RDT_BALANCED_BIDIRECTIONAL($q_I, q_G$)
    $T_a$.init($q_I$); $T_b$.init($q_G$);
    for $i = 1$ to $K$ do
        $q_n \leftarrow$ NEAREST($S_a, \alpha(i)$);
        $q_s \leftarrow$ STOPPING-CONFIGURATION($q_n$,$\alpha(i)$);
        if $q_s \neq q_n$ then
            $T_a$.add_vertex($q_s$);
            $T_a$.add_edge($q_n, q_s$);
            $q'_n \leftarrow$ NEAREST($S_b, q_s$);
            $q'_s \leftarrow$ STOPPING-CONFIGURATION($q'_n$,$q_s$);
            if $q'_s \neq q'_n$ then
                $T_b$.add_vertex($q'_s$);
                $T_b$.add_edge($q'_n, q'_s$);
            if $q'_s = q_s$ then return SOLUTION;
        if $|T_b| > |T_a|$ then SWAP($T_a, T_b$);
    return FAILURE
```

Figure 5.24: A bidirectional RDT-based planner.

1. **Algorithm**

   In this method we grow the map both from the start vertex and the end vertex. I used two HashMaps to represent each tree (The graph is made from two trees in this case). After initializing the maps we choose a random configuration in the space, a_i and find the nearest neighbor, q_n, to a_i. Next we move to a random direction from q_n for a certain duration until we get to a new configuration q_s. In order to force the graphs to grow toward each other instead of using another random configuration and finding the nearest neighbor to it from the second tree we will use q_s and find the nearest neighbor to q_s form the second tree. Using this trick we will force the trees to grow toward each other.

   This method performs best when the trees are close to balanced. In order to make sure the start and goal trees are growing in a balanced way, after each iteration of expansion we will check the size of the trees and swap the reference of the trees if one tree's size is larger than the other one. Using this trick makes sure that each tree has a chance of being the one with random configuration expansion that comes from making a new a_i at each iteration.

Notice that the reason for unbalanced growth of trees lies with the fact that there are going to be obstacles on the map which will stop a configuration to become part of a tree. I used the code below to swap the trees:

```
private void swapTrees(Map<Vector, Edge> t1, Map<Vector, Edge> t2){
  Map<Vector, Edge>  tempMap = t1;
  t1 = t2;
  t2 = tempMap;
}
```

2. **Implementation** I followed the psudocode above for implementing the algorithm. Most of the implementation was straight forward as it was similar to the code for single tree implementation of RRT that we used in motion planing assignment. The tricky part of the implementation was to decide what to do when the graphs meet each other. Since we are using doubles to represent a configuration (or vertex) it is not feasible to check if the two newly added vertices in the two trees are the same configuration hence I used a variable **epsilon** and if two vertices from the two trees were a distance of epsilon or closer apart from each other I would consider them connected and would connect them with a link (trajectory) to each other. At that point there is a decision to be made. We can either continue to expand the trees or we could stop after finding at least one connection point!

   In order to deal with connecting the two trees I have made a method called **connectTrees()**:

```
  private boolean connectTrees(Vector qs, Map<Vector, Edge> tree,
  Vector qs_hat, Map<Vector, Edge> tree_hat){
  //make trajectory between the two vertices(make one!)
  Vector control1 = getNormDirection(qs, qs_hat); //direction
  double len = getMagnitude(qs, qs_hat);
  Trajectory t1 = new Trajectory(control1,(len/DEFAULT_DELTA)*DEFAULT_DELTA);
  //check if that trajectory is valid (use isValidMotion)
  if(getEnvironment().isValidMotion(getRobot(),qs,t1,RESOLUTION)) {//if the
  trajectory is validMotion
```

```
 //make edges out of the vertices and trajectories
 Edge e1 = new Edge(qs, t1); //Edge form qs to qs_hat
 if(!parents.containsKey(qs_hat)){
 parents.put(qs_hat, e1);
 }else{
 return false;
 }
 connection = new Pair<>(qs_hat, e1);
 connected = true;
 System.out.println("Connected!");
 return true;
}else{
 return false;
}
}
```

The variable **"connection"** in the code above is the connection point between the two trees. It is initially art of the second tree, however, it is also added to the first tree (the start tree) in the connectTrees() method. After the connection is made we are left with two trees that are connected to each other at one leaf. Notice that at this point the trajectories from the goal tree are moving from goal to the connection point which is the opposite of the trajectories that we need. Hence I made two changes to overcome this. First in the **backchain()** method I added the trajectories in reverse compared to the start tree as shown below:

```
 ...
if(useBidirectionalMap) {
for(Vector v=currentConfig; v!=null; v=parents2.get(v).parentVector){ //for
//the second half starting at the connection vertex
      result.addLast(v);}
for(Vector v=currentConfig; v!=null; v=parents.get(v).parentVector){// for
//the first half starting at the connection vertex
      result.addFirst(v);}
}else{
...
```

The next change was in the **convertToTrajectory()** method. While appending the trajectories of the vertices that were returned from **backchain()** the trajectories from the goal tree had to be reversed in direction:

```
...
if(firstHalf){//if using the trajectories form the first half of the path
    if(parents.get(v).parentTrajectory != null){
        result.append(parents.get(v).parentTrajectory);
    }
}else {//using trajectories from the second half of the path
    if (parents2.get(v).parentTrajectory != null) {
        result.append(reverseTrajectory(parents2.get(v).parentTrajectory));
    }
}
...
```

3. **result** Figures below show the Balanced-Bidirectional-RRT in action. For the figures below i have used the following parameters:

   DEAFULT-DELTA = 0.1

   epsilon = 0.1

   default growth size = 10000 (Note that the expansion stops after finding the first connection (8000 using these parameters))

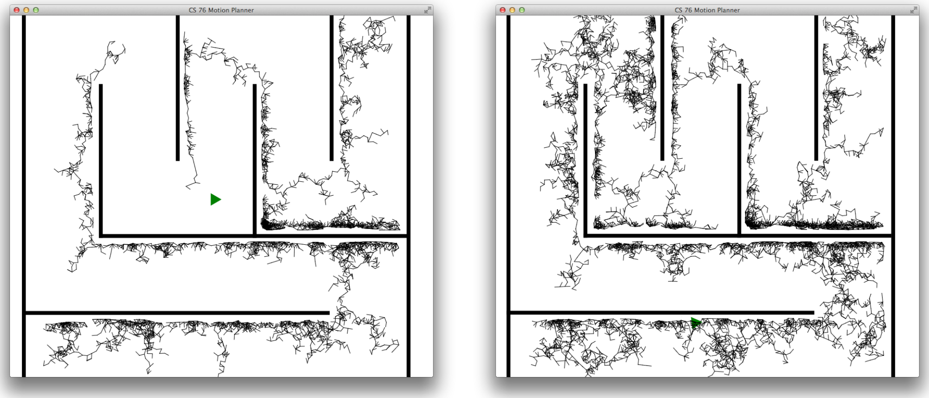


Figure 2: There are two trees starting from start and goal vertices (Left) the two trees meeting each other and creating a connection point (Right)