Assignment 3 Record Log/Division of Labor

By: Team Immortal

Division of Labor:

Robert Pierce: Worked primarily on implementation for ExtendTree and ExtendRandom.

Sara Saad: Also worked on ExtendTree and ExtendRandom for some part; worked primarily on ExtendRRT and ExtendEST

Josh Lilly: Worked on ExtendTree and ExtendRandom

Joe Brock: Worked on the Assignment 3 Record Log, pseudocode for explaining our methodology, and helped with the coding and debugging of ExtendTree. ExtendMyApproach, and ExtendRandom.

Zifu Wang: Worked on ExtendMyApproach and helped debug ExtendRandom

Our Approaches to the Problems provided:

During this project, we had a variety of different methods that we used to implement our Tree structure, our Random function, our RRT, our EST, and our own approach. Here is a brief description of the steps that we took in order to get the most of our functions, as well as the pseudocode it took to create them:

1. Our Tree Extension Function

* A Basic Explanation:

For our Tree Extension function, the main purpose of the function was so as to determine two things: 1) That the nodes that we were sampling were not within any obstacles, and that 2) The edges that we had planned out where also not within any obstacles.

The way that we implemented included the following steps. First, we took our sample state, and then determined how far our current state was from where we were previously. If our radius was greater than our robot’s radius, or if our state is not valid, we would just return nothing. This was so that way we could stop it in its track in the case that there was some level of intersection, and so that way we didn’t add that incorrect vertex. If it had to make it past our first check, however, then we would iterate along the line and check to see if there were any obstacles on the way to our goal. In the case that there weren’t any obstacles, we would simply add our new vertex to our possible paths.

* Our Pseudo-Code:

> Get the params for vid and our step size

> Set our robot’s state to hypothetically be at where our step size is

> Run a for loop for all of the vertices inside our array of possible vertices

+ If the distance < our Robot’s radius, then we know that our step size is too small and therefore we need to return nothing.

+ If that’s not the case, we continue with our loop and check to see if other vertices within our list are equidistance or are > our robot’s radius

> Get the X and Y value of our random vertex (which is considered our previous node); determine our magnitude from the following formula: sqrt((x^2)+(y^2))

> Check the possible states on the line that we’ve just created to ensure that there are no obstacles within our edge. In the moment that we have an obstacle within our edge, we must return null (because we can no longer use it!)

> If we’ve reached our goal, we ask our robot that if we have reached our goal

+If we have, we must tell the robot that we have found our optimal path

+If not, we just add the Vertex to our list of possible vertices

1. Our Random Function

* A Basic Explanation:

The way our Random function is supposed to work is that, given a random state generated by our code, we should move our position based on that state. To accomplish that, our first step is to first create a state in which it’s within our radius. Then, what we do is that we check to see if this is our first vertex or not. If this is our first, then we call extend tree and extend the tree. However, if not, then we create a random vertex for our possibility space based on our max number of vertices available, and then call extend tree with.

* Our Pseudo-Code:

> Create a sample state

> Have our robot then assert that this is now a possible position on our tree

> In the case that our max != 0, create a random vertex index for our Vertex array

+ In the case that max == 0, don’t create a random vertex just yet

>Extend the tree until we reach goal

1. Our RRT

* A Basic Explanation:

For our RRT, we designed is so that way it would be centered more around optimality instead of anything else as opposed to something like EST. The idea behind it is that we want to have a surefire way of getting the most optimal path to our goal, and therefore the way that we implement that is by skipping nodes that are proven to be not optimal for getting us to that goal. First we would have to generate a random sample and then find the closest index possible. Then, what we do is that we get the smallest possible distance between our random sample and other samples that we input into our code. Then, we extend our code to tree and run until we hit goal.

* Our Pseudo-Code:

> Create Our Sample

> Get the Closest Vertex to our Sample

> Get the distance between our closest Vertex

> Run a for loop for other possible vertices

+ If any other vertex happens to be smaller than our current distance, our smallest distance comes to that and our closest vertex changes

> Then, extend the tree and repeat until goal is found

1. Our EST

* A Basic Explanation:

Since EST is more exploration-based than RRT (i.e. it relies on probability distribution and uses weights as a means to find a possible path), what we do with this tree is that first we first make sure that we have a random sample that we can choose from. Then, what we do is that we create a value called “weight” and sort through all possible vertices. Within our loop, we’ll continuously add to our weight based on the following equation: (1/(vertex’s children)^2). When we’re done with that, we generate a pseudo-random number between 0 and our weight, and then create another for loop where we do the exact same thing as we did in the previous one, but instead we check when our weight is greater than rand so that we can generate our random index and feed to our ExtendTree(vid,samples).

* Our Pseudo-Code:

> generate random sample, random index and weight (which equals zero, initially)

> run a loop through every vertex

+Here, we continuously add (1/(Vertex’s children num)^2) to our weight

> generate a random number afterwards (known as rand)

> run another for-loop through every vertex

+Do exactly what we did in the first, but this time, if our weight is

greater than our random number, we set our new random index to

our current position within the vertices, and then break

> extend the tree and repeat until goal is found

1. Our Approach

* A Basic Explanation:

For ExtendMyApproach, the way we approached the problem was that we decided we wanted to utilize the closest possible point to our goal (and, in the case that we couldn’t find it, select a random point and then build a path that could possibly get us closer to the goal). To do this, what we did was that we had to create a loop that would go through all possible vertices and then pick the closest vertex to the goal. In the event that we couldn’t go any further to the goal based on what we consider the closest point, then we decide to pick a random node and then determine whether or not we can move forward based on what is the most optimal solution for the time being given our former optimal solution is now impossible.

* Advantages/Disadvantages:

Some advantages to this is that it allows us to head straight to goal, sometimes even without having to do all of the possible optimizations that would make it difficult at first glance. However, given more objects encased in a spiral, one of the disadvantages to this approach is that though there is some randomness, there is the possibility our approach could fail on the basis that our “random points” are really more focused on optimality than practicality (i.e. since we’re so concerned about getting to the goal, we may have a longer computational time because the more obstacles that we have, the more optimal paths we’ll have to run through and eliminate as time goes on dependent). Granted, this is all based on how objects are placed within a given space, and our method has been proven to be the most optimal in areas where a multitude of objects are more or less scattered around.

Our Pseudo-Code:

> First, determine the minimum distance between the randomly generated sample you have and your first vertex

> Then, check to see the distance between a possible vertex and your goal’s vertex.

+In the case that your vertex for the goal is smaller than the vertex for your initial data, change your minDistance and your closest Vid.

>If you’ve had to change your closest Vid, then extend the tree (ignore bottom steps) and determine if you had added any new nodes

>If not, then generate a pseudorandom number

+If the dice result doesn’t equal 9, then move towards a random state

+If it does equal 9, head over to the goal state

>Then, pick random vertex, and extend the tree again