MEAM 520 LAB4

Vanshil Shah, Divyanshu Sahu

Methods

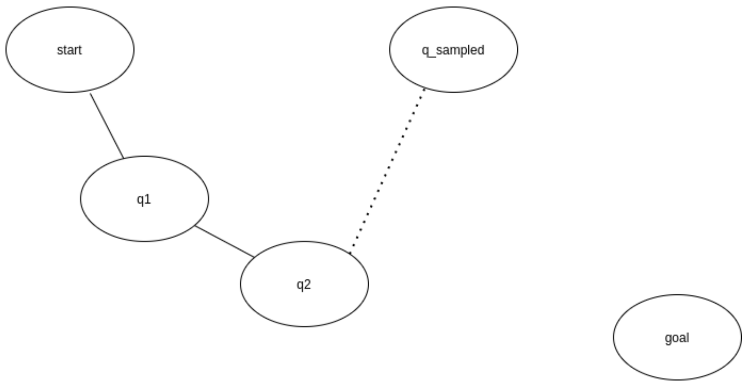
Overview of our implementation:

The RRT implementation consists of using the tree data structure for the path planning problem. We sample random configuration in the configuration space and check for the collision as per the collision criteria which will be explained below. If the sampled point obeys the collision criteria, we add the point to the tree otherwise we discard the point and start over. If the added point in the tree is within a tolerance radius of the goal configuration, the added node is connected to the goal after doing the collision criteria check again. If this check is passed, the path is complete and we calculate the path by joining all the points in the tree. The pseudo code of our implementation is as follows.

```
$T_{start} = (q_{start}, \phi)$
while $i \leq n_{iter}$ do
    $q \leftarrow q_{random}$
    $q_a \leftarrow closestvertex(T_{start})$
    if $qq_a$ not collided then
        $Tree \leftarrow Add(qq_a)$
        if $q_a'$ within defined tolerance of goal then
            if $q_a goal$ not collided then
                $Tree \leftarrow Add(qq_{goal})$ break
            end if
        end if
    end if
end while
```

Code implementation details:

Each node in the tree is a struct which contains the configuration of that node along with the index of the parent (in the array of tree nodes) it is attached to. This implementation helped us a lot to keep a track of the tree nodes and their corresponding parent. We initialise the tree data structure by initialising the start position and adding it to the tree array. Note that the start point will not have a parent.



Then we randomly sample from the configuration space of the robot bounded by joint limits. Sampling points in the configuration space will be more efficient as compared to sampling the points in the workspace. This is because if sampled in the workspace, the inverse kinematics would be used to calculate the corresponding configuration which might have multiple solutions as the manipulator is a redundant one.

Sampling in configuration space on the other hand will aid in calculating a unique forward kinematic solution which would give the position of the end effector in the workspace.

Now, for this sampled point we find the nearest neighbour node in the existing tree according to euclidean distance. We then connect the line from the goal to the sampled point and this path is then checked for collision.

The collision criteria:

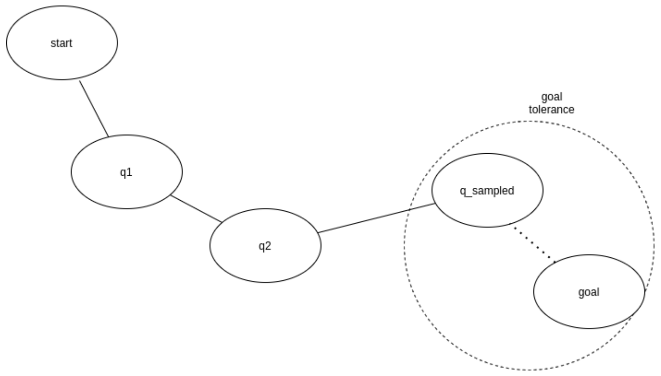
We are performing collision checking in the workspace. Doing sampling iteratively in the configuration space and mapping the sampled configuration in the workspace helps in collision checking. This is because mapping the obstacle to the configuration space which is 7 dimensional space would be computationally expensive as there exists infinite many configurations leading to collisions.

A straight line in configuration space doesn't necessarily imply a straight line in the workspace of the robot. Hence, we are dividing the line connected between q_sampled and the nearest neighbour in the tree into multiple small segments which can be approximated as a line movement in workspace. After this, we check for collisions of each segment using the detect collision functionality for each of the segmented line segments and the box. Note that we have added inflation to the obstacle box to account for the volume of the link and to be conservative in giving our solution.

Goal tolerance check:

To make our algorithm converge, we have implemented goal tolerance to make the final connection in our RRT algorithm. After passing the collision check criteria shown above, the node is added to the tree. Now, if this sampled node is near the goal and within the tolerance limit set by us, then we connect the node to the goal and check for collision using the same collision criteria as mentioned above again. If the connection is collision free, our algorithm converges.

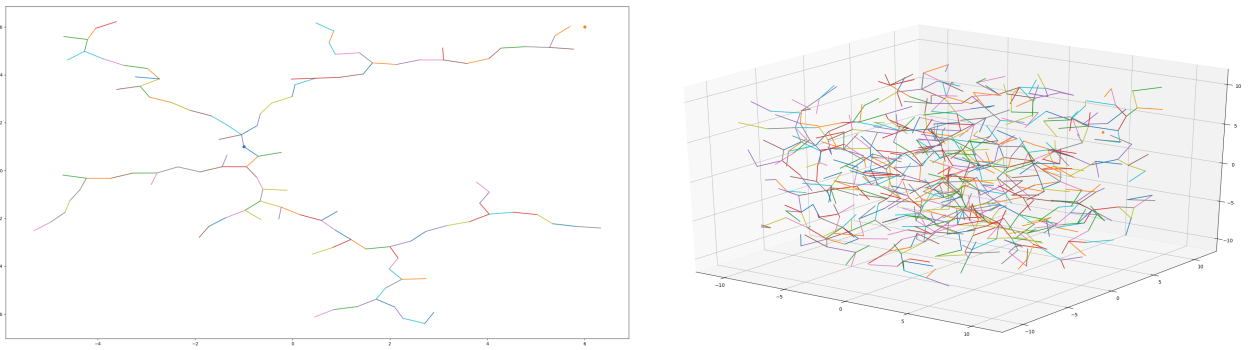
After our algorithm converges, we backtrace the nodes starting from the goal and going to the start point. This back tracing became easy as we had kept track of all the parents in the optimal path.



Evaluation

To check the validity of our RRT we first visualised our implementation in 2D and 3D space without obstacles before moving directly to the 7d space. For this, we simply assumed our configuration space being 2D and 3D and plotted the output path. This approach helped us a lot in removing the initial bugs of our vanilla RRT implementation and understanding the effects of different parameters in the RRT code on the final path implementation.

The following graphs show the result of our implementation of the vanilla RRT without obstacles. We have visualized the RRT implementation in both 3D and 2D space the results of which are shown in the following figure below.



2D and 3D implementation of vanilla RRT

To evaluate the performance of our algorithm, we tested the RRT and A-Star algorithm with the following start, goal and environment variables (obstacle coordinates) and tested the time taken to find the path for both methods. We ran each test for each of the algorithms 3 times and calculated the average time for each planner. The results are as follows:

1) start = np.array([0, -1, 0, -2, 0, 1.57, 0])
   goal = np.array([-1.2, 1.57 , 1.57, -2.07, -1.57, 1.57, 0.7])
   Average execution time in seconds RRT: 32.694764137268066 seconds.
   Average execution time in seconds A-Star: 153.75534176826477 seconds.
   Obstacle block coordinates .15, -.300, 0.496825 , .45, .300, 0.503175.

2) start = np.array([0, -1.5, 0.2, -2, 0, 1.57, 0])
   goal = np.array([-0.2, 0.57 , 0.57, -0.07, -1.57, 1.57, 0.7])
   Average execution time time in seconds for ASTAR: 9.78854489326477
   Average execution time in seconds RRT: 3.7123324871063232
   Obstacle block coordinates .18 0.19665 -0.25 0.69 0.20335 0.65
   Obstacle block coordinates .18 -0.20335 -0.25 0.69 -0.19665 0.65

3) start = np.array([0, -1.5, 0.2, -2, 0, 1.57, 0])
   goal = np.array([-0.2, 0.57 , 0.57, -0.07, -1.57, 1.57, 0.7])
   Average execution time in seconds ASTAR: 9.355504035949707
   Average execution time in seconds RRT: 3.815410614013672
   Obstacle block coordinates .18 0.19665 -0.25 0.69 0.10335 0.55
   Obstacle block coordinates .18 -0.20335 -0.25 0.49 -0.19665 0.65

4) start = np.array([1, -1.5, 0.2, -2, 0, 1.57, 0])
   goal = np.array([-1.2, 1.57 , 0.57, -1.07, -1.57, 1.57, 1.7])
   Average execution time in seconds ASTAR: 19.592761754989624
   Average execution time in seconds RRT: 7.63759708404541
   Obstacle block coordinates 0.15 -0.2 0.0 0.1 0.0 0.12

5) start = np.array([1.5, -1.0, 1.2, -1, 0, 1.57, 0])

```
goal =   np.array([-1.2, 1.57 , 0.57, -1.07, -1.57, 1.57, 1.7])
Average execution time in seconds RRT: 20.05246376991272
Average execution time in seconds ASTAR: 42.76940369606018
Obstacle block coordinates 0.15 -0.2 0.0 0.1 0.0 0.12
```

How often does your path planner succeed?

- The RRT planner will take a long time to find a path when the goal is very close to the inflated obstacle. When the goal point is inside or on the edge of the inflated obstacle, the algorithm fails.

How long does it take for your planner to find a path (running time)?

- In comparison to A-Star, the RRT algorithm is much faster and takes less time to plan for the same start, goal positions and obstacles. The average runtime for both RRT and A-Star is shown above.

When your planner finds a path, is the path the same over multiple runs?

- In RRT the sampling method is random, hence every time a different path is returned and the amount of time it takes to find a path also changes. Hence, the path is different every time we are planning a path with the same start, goal positions and obstacles. But for A*, as the algorithm tries to find an optimal path, the path remains the same for a given heuristic and given environment conditions(start, goal and obstacles).

How do the answers to these questions change for different environments or variations in your implementation?

- The time taken for the RRT to converge is subject to how conservative our implementation is and how difficult it is to reach the goal position with respect to the placement of obstacles. If the goal lies in a very narrow space, between obstacles, the probability of sampling a point in that space near the goal becomes very less. Hence finding a path to the goal becomes difficult. Moreover increasing the precision of our collision checking algorithm, by dividing it into more line segments results in more time. A star algorithm will take more time when we increase the number of grids of the occupancy map that the algorithm searches in.

Analysis

What kinds of environments/situations did your planner work well for? What kinds of environments/situations was it bad at?

- RRT wont work well or converge fast when the goal position is near an obstacle. Moreover, in our implementation of RRT, we have approximated the movement in the configuration space to be the same as that in the workspace. Moving in a straight line in the 7 dimensional configuration space will not be the same as moving in the workspace. Although we have segmented the line line connecting the line connecting the nearest neighbour to the sampled point to decrease the extent of this effect, it still is not perfect and may lead to collisions if not segmented appropriately.

- Lastly, we have set the inflation radius conservatively, this is a good consideration for the volume of the robot. However, we are approximating the inflation radius which may cause the goal to end up inside the obstacle or near the edge which may cause the algorithm to not converge or converge after a very long time.

What issues did you run into or what differences were there between your expected and experimental performance?

- We first included goal tolerance as a metric for convergence of the RRT algorithm. This led to a lot of convergence issues as the RRT algorithm randomly sampled configurations and it was very unlikely that the sampled configurations would be within that tolerance.

- Moreover, we had also considered a step size which would restrict the size of movement towards the sampled goal so that the sampled configurations wouldn't be so sparse that it would make our collision checking faulty. This was an effective way to restrict the maximum movement of the in the sampled direction which made the collision checking effective as the approximation of a linear movement in 7 dimensional space would be a plausible one. However, this highly restricted the movement of the sampled configurations in all the directions and increased the number of iterations the algorithm had to take before the convergence happened.

- To overcome this, we implemented the segmentation part of the algorithm which did both a good job of moving a significant amount in all directions as well as had good collision avoidance capabilities.

- Also, adding inflation radius was an important consideration in our algorithm. This made the checking of obstacles with collision avoidance effective. However, this parameter needed to be carefully tuned as a huge inflation radius made the goal point be inside or nearby which made the convergence of our algorithm difficult. Also, a very small value would be a bad assumption of the link volume of the robot and may lead to collision

What changes would you make to your implementation if you had more time with the lab?

- We would improve the convergence time it takes for the RRT algorithm by implementing a bidirectional search algorithm or an implementation of RRT star which finds an optimal path. Moreover, biased sampling would also help improve the performance of this algorithm. We will also try to implement a trajectory smoothing algorithm such that velocity and acceleration of links to reach that point in trajectory can also be controlled.

- Moreover, extra collision checking algorithms would be added to account for self-collision and collision between the two joints of the robot.

What differences do you notice between the grid based planner (A*) and the random search (RRT)?

- A-Star algorithm is a graph based planner which discretises the entire configuration space and tries to find an optimal path . The implementation of this algorithm will give an optimal path given our choice of heuristic is correct. However, this algorithm is computationally expensive if the grid' resolution is very high or configuration space is high dimensional.

- On the other hand, RRT is a probabilistic sampling based planner which overcomes this problem of high dimensional sampling. This algorithm guarantees a path but not an optimal one. The convergence time however, is highly efficient as compared to that of the A-Star algorithm. Hence the main trade off between the two algorithms is optimality vs the convergence time.