Robotics Assignment 5

10 February 2019

Group (Thu_B)

Alessandro Amici	404920
Kai Uwe Jeggle	401681
Sergio Lezama Ruvalcaba	404880
Jiaqiao Peng	406038

Implementation Table

Group Members	1. Exhausted nodes	2. Sampling near collisions	3.1 Gaussian sampling	3.2 Bridge sampling
Alessandro		x		
Kai	x			
Sergio			x	x
Jiaqiao Peng				

Group Members	4. Dynamic-Domain RRT	5. 1 and 2 joined
Alessandro		
Kai		x
Sergio		
Jiaqiao Peng	x	

1. Description of the algorithms

Extension 1: Exhausted Nodes

Count for each node the number of times its expansion fails

1. Approach
   - when counter n reaches a given threshold, the node is no longer considered in the nearest neighbour selection.
   - After testing different values for the threshold, we found that the algorithms runs fastest with a threshold of 4.
2. Approach
   - penalize nodes with a higher counter n, but still consider them in the nearest neighbour selection
   - calculate history-based weighting H for every node and choose node with lowest H as nearest neighbour

- disadvantage of this approach is, that all vertices need to be looped through twice for every nearest neighbour selection. The first loop for calculating n$_{min}, n_{max}, d_{min}, d_{max}, Secondlooptoactuallycalculate H$

$$H(x^j, x^{rand}; d) = \frac{d(x^j, x^{rand}) - d_{min}}{d_{max} - d_{min}} + \frac{n^j - n_{min}}{n_{max} - n_{min}}$$

Extension 2: Sampling near collisions

In this extension we implemented a different approach to sampling. The idea is that it is not worth it to have a lot of samples in the free space. It is much more interesting to sample near obstacles in order to try to surround them with sample and therefore, connect two previously separated free spaces.

The default implemented version of the RRT is maybe good in overcoming obstacles in general, but it performs very badly in the case of a narrow passage, because it uses random sampling. Thus, it is not very likely that it would sample very close or exactly at the exits of a narrow passage.

Algorithm 1 Sampling near collisions

```
1: if collisionCounter > perseverance then
2:     sigma = randomSigma
3:     lastCollisionPoint = random()
4:     collisionCounter = 0
5: else
6:     collisionCounter ++
7: end if
8: median = lastCollisionPoint;
9: return sampleGaussian(sigma, median)
```

Algorithm 2 Updating collision point

```
1: if collisionHappened then
2:     sigma = collisionSigma
3:     lastCollisionPoint = lastConnectedNode
4: end if
```

This code presents three different hyper-parameters that could be further tuned to obtain the best possible result:

perseverance

Indicates the number of samples that we want to generate near the same obstacle. Every *perseverance* samples it samples randomly in order to change the target obstacle. The idea is that we want to sample near obstacles, but we may end up sampling near an obstacle that is not the one that impede the tree to reach the goal (or in our case the other tree). Therefore, it is smart to have a way to change the focus to explore another area.

randomSigma

Indicates the value of sigma when random sampling. For simplicity we decide to use a gaussian sampler with a high sigma in order to generate approximately uniform distributed samples.

collisionSigma

Indicates the value of sigma to use when sampling near a collision point. It must be relatively small in order to sample close to the collision point. A too small value is not wanted because it would lead in a longer time in exploring the same area.

The final result is very good. This approach is much faster than the default one.

Extension 3: Gaussian Sampling and Bridge-Test Sampling

For this extension we implemented two different sampling methods, instead of using the uniform sampling. Extension 3.1 uses a Gaussian sampling method and extension 3.2 uses the bridge-test sampling method.

Extension 3.1 generates two samples, the first one q_1 is generated uniformly at random, and the second one q_2 is generated according to a Gaussian with a mean of q_1. Afterwards both samples are compared to determinate if one of them lies in the observed space, and the one that lies in the free space is kept as a vertex.

Extension 3.2 is also a Gaussian sampling strategy, but here three samples along the same line are used. If the first and last samples lie in the observed space and the sample in the middle is in the free space, then the sample in the middle is kept as a vertex. This bridge-test help to determine if a vertex can be located in a thin corridor. The idea is to improve the coverage of the free space.

After implementing both extensions and comparing them to the default uniform sampling, we can observe that the Gaussian sampling performs better in average, but it is not a great improvement. For the bridge-test sampling we observe that it is slightly worse than the default and the Gaussian, nevertheless it is not a considerable difference.

Extension 4: Dynamic-Domain RRT

Algorithm 3 BUILD DD RRT(q$_{init}$)

```
1: $\tau.init(q_{init})$
2: for $k = 1$ to $K$ do
3:     repeat
4:         $q_{rand}$ ←RANDOM_CONFIG();
5:         $q_{near}$ ←NEAREST_NEIGHBOR($q_{near}, \tau$);
6:     until $dist(q_{near}, q_{rand}) < q_{near}$.radius
7:     if CONNECT($\tau, q_{rand}, q_{near}, q_{new}$) then
8:         $q_{new}$.radius = $\infty$
9:         $\tau$.add$_v ertex$(q$_{new}$);
10:        $\tau$.add$_e dge$(q$_{near}, q_{new}$);
11:    else
12:        $q_{near}$.radius = $R$;
13:    end if
14: end for
15: return $\tau$
```

The pseudocode of the DD-RRT algorithm is shown above. At the beginning of the exploration of the tree, all the points are considered to be nonboundary. As soon as the expansion of a given node fails, it becomes a boundary node. This corresponds to the lines 11-13 in the code. The radius field of this node is then updated to R.

Compared with the default planner, DD-RRT achieved less avg Nodes and avg Queries but higher avg time. The DD-RRT algorithm is an algorithm especially efficient for problems with complex geometries where the collision tests are expensive. It is not really suitable in that situation. Besides, there is a radius parameter. The bigger the radius, the stronger the exploration, which also causes the detriment of the tree refinement. Adaptive tuning may improve the result.

Actually, the author proposed a variant algorithm to adapt the radius of each of the nodes independently during the search process. Due to the time constraints, only origin DD-RRT was implemented.

2. Performance evaluation

Planner_Name	defualt	(REV)	1. Exhausted nodes	2. Sampling near collisions	2. (REV)
avgT	22398.83	20316.85	40900.62	2135.52	3688.43
stdT	11029.82	20466.30	47510.19	2382.74	5026.61
avgNodes	12495.75	10163.42	47510.19	1748.77	2620.35
avgQueries	578095.25	509572.00	706980.30	66117.42	102412.60

Planner_Name	3.1 Gaussian sampling	3.1 (REV)	3.2 Bridge sampling	3.2 (REV)
avgT	18994.21	19551.80	33885.19	15130.51
stdT	13054.23	12175.58	31050.90	7987.79
avgNodes	10164.09	10388.10	14467.10	8477.90
avgQueries	514227.64	521829.50	698880.20	439399.20

Planner_Name	4.Dynamic-domain RRT	4. (REV)	5. 1.1 and 2 joined	5. (REV)
avgT	27127.17	26797.11	1758.91	1897.79
stdT	25408.00	22938.13	2375.92	3050.47
avgNodes	11862.82	11988.20	1779.85	1813.95
avgQueries	580248.09	585146.90	56282.15	57270.95

As a conclusion, we noticed that the sampling strategy is the most relevant factor that makes our extensions better than the default one. Nevertheless, our best implementation contains some hyper-parameter that can change drastically the performance of the algorithm. As noticed from the beginning of the course, tuning is a key step to make a solution working correctly.