Technische Universität Berlin
Fakulät IV
Institut für Technische Informatik und Mikroelektronik
Fachgebiet Robotics and Biology Laboratory

Robotics
Prof. Dr. Oliver Brock
WS 2019/20

Assignment 5

Marvin Janitschke 371493
Benjamin István Berta 415348
Janesh Alladi 415349

09. February 2020

Contents

1 Standard RRTConCon Algorithm

In table 1 are the average parameters of the standard RRTConCon algorithm from the Robotics library shown. The task of this assignment was mainly, to improve those parameters. Most of our modifications were therefore tested on there contribution to the improvements. We use these for comparison.

Table 1: Execution Times and Node numbers - Standard RRTConCon

	RRTConCon	RRTConCon (reversed)
avgT	65474.3	83320.92
stdT	40694	50516
avgNodes	10616	12602.4
avgQueries	535438	623072

2 Modification A - Avoidance of exhaustive nodes

The first extension we tried was based on [1]. The idea of that extension is basically, to try to avoid exhaustive nodes in the RRT. Exhaustive nodes are vertices in the tree, which can not connect to newly sampled configurations. These are often close to the border of feasible and unfeasible regions in the configuration space and therefore often selected for expansion or connection [1]. By avoiding them, one can theoretically decrease the number of nodes in the tree as well as finding a fast solution. In the following three subsections, we tried different strategies to avoid these nodes. All feasible extensions to avoid exhaustive nodes were tested and the recorded parameter are shown in table 2.

Table 2: Execution Times and Node numbers - Avoid exhaustive nodes

	YourPlanner A	YourPlanner A (reversed)
avgT	72420	80415.5
stdT	39865	36974
avgNodes	10410	11121.2
avgQueries	546290	601260

One can see, that the improvements are not very big. For the normal problem the results are mainly the same (or even slightly worse) but for the reversed one, the results with the improvements of A are slightly better. This is because in the given scenario is not too much space for exhaustive nodes, as much of the space is open with only a few obstacles in it.

2.1 Limit Connection Attempts

The first extension was to limit the connection attempts of each node. We therefore implemented a counter to count every time, a node tried to connect to its nearest neighbor, but was not successful. If this counter has reached a predefined level, the respective node is removed from the tree and is not used for any extensions anymore. In listing 2.1 is the respective code snippet shown, which is implemented in the function RRTConCon-Connect.

```
1  //Check if vertex could have been connected
2  if (check_vert == NULL){
3      //Check if nearest vertex is exhaustive, but do not remove
           vertices, already connected
4      if((tree[p.first].counter >=
           EX_LIM)&&(tree[p.first].already_connected == false))  {
5        //Remove Vertex
6        RrtConConBase::Vertex v = p.first;
7        remove_vertex(v, tree);
8        std::cout<<"removed vertex"<<std::endl;
9        return NULL;
10       }
11       //Increase counter
12     else { tree[p.first].counter++;}
13 }
14 //Vertex was sucessfully expanded
15 else{
16     tree[p.first].counter = 0;
17     tree[p.first].already_connected = true;
18 }
```

For the implementation of that extension we had to expand the attributes of the RRTConCon Vertexbundle. We added the two variables counter (int) and already-connected(bool). We also observed, that the hugest improvement is gained, when the limit is set to 3.

2.2 k-Nearest Neighbor Search

The second idea to avoid exhaustive nodes was to use not only the most nearest neighbor of the newly chosen configuration, but to select one of the k-nearest neighbors. Theoretically this will improve the space coverage, because it also gives a chance to nodes, which are not

in the same tree [1]. This extension consists of two tasks. The first to find the k-nearest neighbor and the second to select one of these. In listing 2.2 are the two code snippets shown.

```
1  void YourPlanner::find_k_nearest(Tree& tree, const
       ::rl::math::Vector chosen, std::pair<RrtConConBase::Vertex,
       ::rl::math::Real> *k_nearest, int k)
2  {
3     std::pair<RrtConConBase::Vertex, ::rl::math::Real>
          all_vertices [num_vertices(tree)];
4    int counter = 0;
5    for (RrtConConBase::VertexIteratorPair i =
         ::boost::vertices(tree); i.first != i.second; ++i.first)
6    {
7      ::rl::math::Vector tree_q = *tree[*i.first].q;
8      ::rl::math::Real d = calc_distance(chosen, tree_q);
9      all_vertices[counter].first = *i.first;
10     all_vertices[counter].second = d;
11     counter++;
12   }
13
14   //Extract k_nearest and store in array
15   for(int i=0; i<=k; i++){
16   ::rl::math::Real largest_d = 0;
17   for (int j = 0;j<num_vertices(tree);j++)
18     {
19     if(all_vertices[j].second > largest_d){
20       largest_d = all_vertices[j].second;
21       k_nearest[i].first = all_vertices[j].first;
22       k_nearest[i].second = RrtConConBase::model->
23       inverseOfTransformedDistance
24       (all_vertices[j].second);
25       all_vertices[j].second = 0;
26     }
27   }
28   }
29 }
30
31 ...
32
33    RrtConConBase::Vertex check_vert;
```

```
34    Neighbor p(Vertex(), (::std::numeric_limits< ::rl::math::Real
          >::max)());
35    if(K_NEAR_BOOL == 1){
36       //Find k-nearest Neighbor and choose random
37       p.second = 0;
38       int k = 0;
39       if(num_vertices(tree) > K_NEAR+1){
40         k = K_NEAR;
41         std::pair<RrtConConBase::Vertex, ::rl::math::Real>
              k_nearest [k];
42           find_k_nearest(tree, chosen, k_nearest, k);
43           k = rand() % k;
44           p.first = k_nearest[k].first;
45           p.second = (::rl::math::Real) k_nearest[k].second;
46       }
47   }
48   if( p.second == 0){
49     p = RrtConConBase::nearest(tree, chosen);
50     //p.first = nearest.first;
51     //p.second = nearest.second;
52     std::cout<<"K_nearest failed. k:"<<k<<" K_NEAR:"<<K_NEAR<<"
          NumV:"<<num_vertices(tree)<<std::endl;
53   }
54   //Get newly connected(?) Vertex, use random k_nearest neighbor
55   check_vert = RrtConConBase::connect(tree, p, chosen);
```

Unfortunately this extension didn't improve runtime, because the calculation of the k-nearest neighbor and its respective selection was to computational intensive on its own (especially for a huge number of nodes). Additionally, the benefit from selecting a random k-nearest neighbor in the given scenario was not to huge. Because of the few obstacles and already a very dense space confinement, the extension often just refined the space but has not extended the tree to the final desired position. The result was therefore an even more dense tree, which was actually not very close to the end position. Maybe this extension provides a better choice for problems which contain more obstacles and therefore maybe benefit from a denser tree.

2.3 Simplified Task-Adapted Metric

The third idea to avoid exhaustive nodes was to implement a simpler, adapted metric. In the basic RRTConCon algorithm all joints are considered for the calculation of the distance

between two configurations. In [1] it is proposed to use a simpler metric, which only uses the most interesting/important joints for the metric. We therefore tried to exclude two of the 6 joints of the PUMA to see, whether this improves the parameters. In listing 2.3 the respective distance function is shown. In that, we simply replace the joints, which should be excluded, with the value of the other configuration, such that the difference of these two will be zero and does not contribute to the overall distance.

```
1  ::rl::math::Real YourPlanner::calc_distance(const
       ::rl::math::Vector q1, const ::rl::math::Vector q2)
2  {
3      ::rl::math::Real d =0;
4      //Checking if Simplified Task-Adapted Metric should be
           applied
5      if(STAM == 1){ //should be applied
6        int jump = 0;
7        ::rl::math::Vector q1_stam = q1;
8        for (::std::size_t l = 0; l < 6; ++l){
9          if(l== SKIP_A || l== SKIP_B){ q1_stam(l) = q2(l); }
10       }
11       //calc the distance
12       d = RrtConConBase::model->transformedDistance(q1_stam, q2);
13         }
14     else {
15       d = RrtConConBase::model->transformedDistance(q2, q1);
16     }
17     return d;
18 }
```

For the given problem, we decided to exclude the last two joints, because they have the smallest link lengths and therefore already don't have a huge contribution to the overall distance. By ignoring these joints the tree was more 'bended' above the wall obstacle, such that the last connection to the second tree could be found faster.

3 Modification B - Dynamic Domain RRT

Dynamic Domain RRT[2] is very similar method to the previously mentioned algorithm which avoids exhaustive nodes. The main idea behind DD RRT is to restrict the sampling domain

of certain nodes in order to prevent unnecessary collision checks.

Each node in the tree receives an attribute which is called radius. This parameter corresponds to the radius of the circular domain of each node. During initialization this parameter is set to infinity (in the implementation: maximum value of the variable), which means that there is no restriction on the given node, and the rules of Voronoi bias applies. When the nearest node fails to connect to the chosen sample the radius of the nearest node is set to a previously defined value. This value is a multiple of the step size and can be configured freely (In the article it is set to 10 times of the step size). From now on this node will be avoided during sampling if there chosen sample is outside of the radius of the nearest node.

This algorithm is particularly suitable for cases where one of the endpoints are "trapped". In our case algorithm did not improve the performance greatly, so we excluded it from the final submission.

```
1  // define radius as multiple of the stepping size
2  double RADIUS = 10 * this->delta;
3
4  ::rl::math::Vector chosen(this->model->getDof());
5
6  while ((::std::chrono::steady_clock::now() - this->time) <
       this->duration) {
7      for (::std::size_t j = 0; j < 2; ++j) {
8
9          ::rl::math::Real distance;
10         Neighbor aNearest;
11
12         do {
13             //Sample a random configuration
14             this->choose(chosen);
15             //Find the nearest neighbour in the tree
16             aNearest = this->nearest(*a, chosen);
17
18             // Calculate distance
19             distance = aNearest.second;
20         } while (distance >= (*a)[aNearest.first].radius);
21
22         //Do a CONNECT step from the nearest neighbour to the
               sample
23         Vertex aConnected = this->connect(*a, aNearest, chosen);
```

```
24
25          //If a new node was inserted the tree
26          if (nullptr != aConnected) {
27              (*a)[aConnected.first].radius = DBL_MAX;
28          } else {
29              restrict
30              (*a)[aNearest.first].radius = RADIUS;
31          }
32      }
33  }
```

4 Modification C - Restricting Sampling Space

Another idea for an extension to improve the solution of the problem was to restrict the space, where the new configurations can be sampled. The intuition of that was to avoid nodes/configuration/samples which are far away from the most probable solution and are therefore not very useful. In this extension we finally agreed on one sphere, which is placed nearly centered above the hole in the obstacle wall and has a predefined radius. If the newly chosen configuration is then inside of this sphere, it will be used, otherwise it has to be sampled again. The respective code snippet is shown in listing 4

```
1      //init spheres
2      end_q = *tree[1][begin[1]].q;
3      end_q(1) += 1;
4
5   ...
6
7   void
8   YourPlanner::choose(::rl::math::Vector& chosen)
9   {
10     if(RSS == 1){
11        ::rl::math::Vector try_chosen;
12        ::rl::math::Real distance_end = 0;
13        ::rl::math::Real distance_right = 0;
14        do{
15          if(GAUSS_SAMPLE==1){
16            try_chosen = this->sampler->generateCollisionFree();
17          }
```

```
18          else {
19            try_chosen = RrtConConBase::sampler->generate();
20          }
21          distance_end = calc_distance(try_chosen, end_q);
22
23        }while((distance_end > RSS_RADIUS_END));
24        chosen = try_chosen;
25     }
26     else{
27    chosen = this->sampler->generateCollisionFree();
28     }
29 }
```

We also added other spheres, which are expanding or reducing the allowed sample space. However, despite of improvements, this has brought, we agreed on not to use these as extensions, because it probably would have been to strongly tailored to the given problem. The sphere we implemented has the final, desired position of the robot (plus some adjustments) as its center. For the reversed problem the had to slightly adjust the sphere. Also, as it was not really possible to find a solution only with the modifications of C, we mixed these with the feasible adjustments made in A. In table 3 are then the parameters shown.

Table 3: Execution Times and Node numbers - Restricting Sampling Space + Avoiding Exhaustive Nodes

	YourPlanner C	YourPlanner C (reversed)
avgT	36949.4	49119
stdT	29162	34994
avgNodes	7030.3	8928.5
avgQueries	169894	202174

As one can see, the combination of the modifications in A and C had a huge impact on all parameters and improved them nearly by the factor 2. By restricting the sample space we also give the path planner a kind of goal bias, that the space the planner has to search for the solution is not that big anymore. Although, we will not use the restriction of the operational space for our final algorithm, because this still is a tailored solution.

5 Modification D - Sampling Strategies

5.1 Gaussian Sampling

Gaussian sampling generates samples that are close to obstacles by generating free samples that are in proximity of a colliding point in the configuration space. Although this algorithm is generally used for PRM, we found that it can also improve the performance of RRT.

The algorithm samples S1 from an uniform distribution and then samples S2 from a gaussian distribution, where S1 is the mean and the variance is arbitrarily chosen. After checking both samples for collision, and only on of them collides, the not colliding samples is returned.

```
1  this->sigma.resize(this->model->getDof());
2      // setting the variance
3      this->sigma << 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f;
4      ::rl::math::Vector q2(this->model->getDof());
5
6      while (true) {
7          // generating an uniform sample
8          ::rl::math::Vector q = this->generate();
9
10         // generating other sample with mean "q"
11         for (::std::size_t i = 0; i < this->model->getDof();
               ++i) {
12             q2(i) = this->gauss() * this->sigma(i) + q(i);
13         }
14
15         this->model->setPosition(q);
16         this->model->updateFrames();
17
18         // checking whether q is colliding
19         if(this->model->isColliding()) {
20
21             this->model->setPosition(q2);
22             this->model->updateFrames();
23
24             // returning q2 if it is not colliding
25             if(!this->model->isColliding()){
26                 return q2;
27             }
```

```
28          } else {
29              // return q if q2 is colliding
30              this->model->setPosition(q2);
31              this->model->updateFrames();
32
33              if(this->model->isColliding()){
34                  return q;
35              }
36          }
37      }
38  }
```

5.2 Bridge Sampling

Bridge sampling is a sampling method which seeks to find samples that are surrounded by obstacles. This should improve the performance of path finding if the configuration space contains narrow passages. In our case, the robot has to maneuver through a hole with it's end-effector, which is a tunnel like shape in the configuration space. Although we predicted that this method would improve the performance, it wasn't very effective.

```
1  this->sigma << 0.1f, 0.1f, 0.1f, 0.1f, 0.1f, 0.1f;
2
3  if (this->rand() > this->ratio) {
4      return this->generate();
5  } else {
6      ::rl::math::Vector q(this->model->getDof());
7      ::rl::math::Vector q3(this->model->getDof());
8
9      while (true) {
10         // generate sample from uniform distribution
11         ::rl::math::Vector q2 = this->generate();
12
13         this->model->setPosition(q2);
14         this->model->updateFrames();
15
16         //check q2 for collision
17         if (this->model->isColliding()) {
18             // generate smaple from gaussian distribution with
                   mean q2
```

```
19              for (::std::size_t i = 0; i < this->model->getDof();
                    ++i) {
20                  q3(i) = this->gauss() * this->sigma(i) + q2(i);
21              }
22
23              this->model->clip(q3);
24
25              this->model->setPosition(q3);
26              this->model->updateFrames();
27
28              // if q3 is colliding, and the middle-point between
                    q2 and q3 is not, return the middle-point
29              if (this->model->isColliding()) {
30                  this->model->interpolate(q2, q3, 0.5f, q);
31
32                  this->model->setPosition(q);
33                  this->model->updateFrames();
34
35                  if (!this->model->isColliding()) {
36                      return q;
37                  }
38              }
39          }
40      }
41  }
```

6 Modification E - Exploration Bias

Another idea of improving the solution is the exploration bias. Normally using Rapidly-exploring Random Tree we sample the nodes randomly. Here we directly select the goal node when we are sampling, so that the tree will grow more towards the goal. By doing this we can reduce the computing time as we target the goal during the sampling. For the sampling to happen this way we have given a threshold to the sampler generator so that if the number of nodes generated reach the threshold, the tree will select the goal as the next node to try and connect to the goal.

```
1   while ((::std::chrono::steady_clock::now() - this->time) <
        this->duration)
2   {
```

Table 4: Execution Times and Node numbers - Exploration Bias

	YourPlanner E	YourPlanner E (reversed)
avgT	63794.35	33375.64
stdT	15559.696	16924.68
avgNodes	10331.5	6259.2
avgQueries	531637.4	336743.7

```
3     //First grow tree a and then try to connect b.
4     //then swap roles: first grow tree b and connect to a.
5
6     //Giving threshold to the number of trees created with the
          samples.
7     for (::std::size_t j = 0; j < 2; ++j)
8     {
9       if(j==0 && this->getNumVertices()%10==0){
10        chosen=*this->goal;
11
12      }
13      if(j==1 && this->getNumVertices()%10==0){
14        chosen=*this->start;
15      }
16      else
17        this->choose(chosen);
18    ...
19  }
```

In code above we have given a threshold so that if the number of nodes reach this threshold then the tree will start to grow towards the goal using goal node as the next point to sample. In the below table we have given the details of the parameters.

7 Final algorithm

For the final algorithm we chose the avoidance of the exhaustive nodes together with gaussian sampling strategy. These modifications have produced the best result together. The results for the 10time repetition experiment are shown in table 5.

All parameters have significantly decreased by a huge factor. The planner finds therefore a very simple and fast solution way faster than the original RRtConCon algorithm. This

Table 5: Execution Times and Node numbers - Final algorithm

	RRTCon-Con	RRTConCon (reversed)	YourPlanner F.	YourPlanner F. (reversed)
avgT	65474.3	83320.92	2095.4	2815.7
stdT	40694	50516	1758.6	1510.5
avgNodes	10616	12602.4	82	94.1
avg-Queries	535438	623072	14961	18357.9

because of our described modifications, which sample only points of interest but also avoid exhaustive nodes. Our other modifications, we haven't used for the final result, may be also relevant for other problems, but didn't contribute huge improvements for the given problem.

8 Submission

Table 6: Workload distribution-1

Student name	Mod. A.1	Mod. A.1 Doc	A.2	A.2 Doc	A.3	A.3 Doc	B	B Doc	C	C Doc
Marvin Janitschke	✗	✗	✗	✗	✗	✗			✗	✗
Benjamin István Berta							✗	✗		
Janesh Alladi										

Table 7: Workload distribution-2

Student name	D.1	D.1 Doc	D.2	D.2 Doc	E	E Doc	Fi-nal	Final Doc
Marvin Janitschke							✗	✗
Benjamin István Berta	✗	✗	✗	✗			✗	✗
Janesh Alladi					✗	✗		

References

[1] J. Cortés, L. Jaillet, and T. Siméon, "Molecular disassembly with rrt-like algorithms", Apr. 2007, pp. 3301–3306. DOI: 10.1109/ROBOT.2007.363982.

[2] A. Yershova, L. Jaillet, T. Siméon, and S. LaValle, "Dynamic-Domain RRTs: Efficient exploration by controlling the sampling domain", May 2005, pp. 3856–3861. DOI: 10.1109/ROBOT.2005.1570709.