Robotics
Assignment 5

Jingsheng Lyu - 398756,
Prathamesh Arun More - 408508,
Lysander Kurt Vincenz Rupp - 359096,
Moritz Duarte Pinheiro-Torres Vogt - 462251

March 8, 2021

Thank you for teaching us these fundamentals of robotics and the extra time given for the revision!

Contents

A Rapidly Exploring Random Trees Extensions

In this assignment the goal is to extend a given motion planning algorithm in order to improve its performance. The given planning algorithm is based on Rapidly Exploring Random Trees (RRT). We obtain a new RRT planning algorithm by changing the nearest-neighbour algorithm, changing the random sampling strategy and by stopping to try to extend exhausted nodes.

A.1 Extension 1: Adaptive Dynamic-Domain RRT

Yershova et al. (2005) discusses the difficulties encountered in sample-based motion planning (e.g. narrow passage problem) and introduces an improvement of the original RRT planner called dynamic-domain RRT [1]. A new variant of the dynamic-domain RRT is introduced by Jaillet et al. (2005). It *iteratively adapts the sampling domain for the Voronoi region of each node during the search process* [2]. An implementation of this adaptive dynamic-domain RRT (ADD-RRT) planner can be found in the Robotics Library[1] which is used in this assignment.

Only a brief explanation of the algorithm is given here. The details are discussed in [2]. The main idea is that the sampling of the random configuration is adapted based on the visible Voronoi region. Since the exact computation of the visible Voronoi diagramm is computationally expensive, the regions are approximated. Here, for every vertex we construct an hypersphere centered at its vertex. During motion planning this hypersphere tries to approximate the visible Voronoi region of its node by dynamically altering its radius. A big radius leads to a vast exploration of the configuration space while a small radius performs refinement.

The pseudo code of the algorithm is shown in Fig. 1. We start by adding the node of the start configuration to the planning tree. As for every newly added node the radius is initialized with infinity. Like in original RRT motion planner we sample a random configuration and identify its nearest neighbour. A sample is discarded if it does not lie within the hypersphere of its nearest neighbour. This sampling procedure is performed until a random sample lies within the hypersphere. After a sample is successfully drawn, the algorithm tries to connect the node q_{rand} to its nearest neighbor q_{near} using a local planner. If the connection procedure succeeds, it returns a new node q_{new} which is added to the tree by connecting it to q_{near}. Based on the result of the connection procedure the radius of the nearest neighbour is either decreased or increased. Then, If the connection succeeds, a finite radius is increased by multiplying the radius with $1 + \alpha$, where $\alpha > 0$. If the connection fails, we decrease the radius. For a finite radius we multiply with $1 - \alpha$. For an infinite radius we set its value to a

[1] `https://github.com/roboticslibrary/rl/blob/master/src/rl/plan/AddRrtConCon.cpp`

predetermined start value $R > 0$.

```
BUILD_ADAPTIVE_DD_RRT($q_{init}$)
    $\mathcal{T}.init(q_{init})$;
    **for** $k = 1$ **to** $K$ **do**
        **repeat**
            $q_{rand} \leftarrow$ RANDOM_CONFIG();
            $q_{near} \leftarrow$ NEAR_NEIGH($q_{rand}, \mathcal{T}, d_{near}$);
        **until** ($d_{near} < q_{near}$.radius)
        **if** $q_{new} \leftarrow$ CONNECT($\mathcal{T}, q_{rand}, q_{near}$)
            $q_{new}$.radius = $\infty$;
            **if** $q_{near}$.radius $\neq \infty$;
                $q_{near}$.radius = $(1 + \alpha) \times q_{near}.radius$;
            $\mathcal{T}$.add_vertex($q_{new}$);
            $\mathcal{T}$.add_edge($q_{near}, q_{new}$);
        **else**
            **if** $q_{near}$.radius = $\infty$
                $q_{near}$.radius = $R$;
            **else**
            $q_{near}$.radius = $(1 - \alpha) \times q_{near}.radius$;
    Return $\mathcal{T}$;
```

Figure 1 | Pseudo-code of the adaptive dynamic-domain RRT algorithm (from [2]).

To incorporate the extension, we add the necessary parameters of the ADD-RRT algorithm as member variables to our planner class:

```
class YourPlanner : public RrtConConBase
{
    ...
    ::rl::math::Real alpha;
    ::rl::math::Real lower; // lower bound of radius
    ::rl::math::Real radius;
    ...
}
```

The actual algorithm is implemented in the method `adaptiveSolve()` of our planner class. Only the significant differences to the `solve()` method of the `RRTConConBase` class are shown here. Below the new sampling strategy is listed in which non-promising samples are discarded.

```
bool YourPlanner::adaptiveSolve()
{
    ...
    do {
        this->choose(chosen);
        aNearest = this->nearest(*a, chosen);
    } while (aNearest.second > (*a)[aNearest.first].radius);
    ...
}
```

The following code snippet shows how the radius is adjusted based on the result of the connection procedure.

```
bool YourPlanner::adaptiveSolve()
{
    ...
    if (NULL != aConnected) // check if successfully connected
    {
        if ((*a)[aNearest.first].radius <
                ::std::numeric_limits<...>::max())
        {
            (*a)[aNearest.first].radius *=
                (1.0f + this->alpha);
        }
    ...
    }
    else
    {
        if ((*a)[aNearest.first].radius <
                ::std::numeric_limits<...>::max()
        {
            (*a)[aNearest.first].radius *=
                (1.0f - this->alpha);
            (*a)[aNearest.first].radius =
                ::std::max(this->lower,
                (*a)[aNearest.first].radius);
        } else {
            (*a)[aNearest.first].radius = this->radius;
        }
    }
    ...
}
```

The method `adaptiveSolve()` is called from the method `solve()` of our planner class. Based on the value of an enum variable either the original `solve()`

method of class `RrtConConBase` is used or the `adaptiveSolve()` method of the ADD-RRT algorithm.

```
bool YourPlanner::solve()
{
    Solver mode = ADD_RRT_CON_CON; // RRT_CON_CON_BASE
    switch(mode)
    {
        case ADD_RRT_CON_CON:
            return this->adaptiveSolve();
        case RRT_CON_CON_BASE:
            return RrtConConBase::solve();
        default:
            return 0;
    }
}
```

When a new vertix is added to the tree, the radius needs to be properly initialized. This is performed in the method `addVertex()` of class `RrtConConBase`:

```
RrtConConBase::Vertex RrtConConBase::addVertex(...)
{
    ...
    tree[v].radius =
        ::std::numeric_limits<::rl::math::Real>::max();
    ...
}
```

A.2 Extension 2: Approximate Nearest Neighbor Selection using Clusters

Finding the nearest neighbor is a common task in many computer science applications. The naive implementation checks the distance to the query point for each point. This results in a linear time complexity $\mathcal{O}(n)$. A more sophisticated way is to change the data structure and use a k-d tree. This reduces the average time complexity to $\mathcal{O}(\log n)$ [3]. In the following, we address a different approach which is based on clustering.

Instead of going through all the nodes, we try to join neighbored nodes into clusters. Whenever a nearest node is searched, we look for the closest centroid of all clusters. Afterwards, the nearest node within the selected cluster is used as an approximation of the nearest neighbor. The following pseudo-code

(*Algorithm 1*) shows the general implementation of the algorithm.

Implementation 1 Approximate Nearest Neighbour Search

```
procedure NEAREST($\mathcal{C}, q$)
    $d_{nearest} \leftarrow \infty$;
    for each $c \in \mathcal{C}$ do
        $d =$ TRANSFORMED_DISTANCE($q, q^{(c)}_{centroid}$);
        if $d < d_{nearest}$ then
            $d_{nearest} \leftarrow d$;
            $c_{nearest} \leftarrow c$;
    $d_{nearest} \leftarrow \infty$;
    for each $q_j \in c_{nearest}$ do
        $d =$ TRANSFORMED_DISTANCE($q, q_j$);
        if $d < d_{nearest}$ then
            $d_{nearest} \leftarrow d$;
            $q_{nearest} \leftarrow q_j$;
    return $q_{nearest}$
```

As new nodes are added during planning, we need to incorporate them into our cluster structure. This is handled by the method `ADD_VERTEX()` (see *Algorithm 2*). For each tree we have a seperated cluster structure (i.e. $\mathcal{C}^A$ and $\mathcal{C}^B$). Each cluster structure $\mathcal{C}^i$ may contain multiple cluster and each cluster may consist of multiple vertices. Based on the current tree, we pick the respective cluster structure $\mathcal{C}^i$, where $i \in \{A, B\}$. In the beginning the cluster structure $\mathcal{C}^i$ is empty. Thus, a new cluster c_{new} is generated, that contains the new node q_{new} we want to add. This cluster is then added to $\mathcal{C}^i$. If the cluster structure is non-empty, we iterate over all clusters in $\mathcal{C}^i$. The process of adding a node is then decided by a hypersphere. Here, each hypersphere is centered around the cluster's centroid and its radius is given by a predefined threshold. For each cluster we check if q_{new} is located within the hypersphere of the respective cluster. When an appropriate cluster is found, q_{new} is inserted to it and we stop iterating. If no appropriate cluster exists, a new cluster c_{new} is generated, that contains the new node q_{new} we want to add. Please note, that the cluster's centroids are not updated when a new node is inserted, i.e. the centroid associated with a cluster is given by the first inserted node.

Implementation 2 Add Vertex

```
procedure ADD_VERTEX($\mathcal{C}, q_{new}$)
    if $\mathcal{C}.empty$ then
        Create new cluster $c_{new}$;
        $c_{new}$.insert_vertex($q_{new}$);
        $\mathcal{C}$.insert_cluster($c_{new}$);
    else
        $inserted \leftarrow false$;
        for each $c \in \mathcal{C}$ do
            $d$ = TRANSFORMED_DISTANCE($q_{new}, q^{(c)}_{centroid}$);
            if $d \leq threshold$ then
                $c$.insert_vertex($q_{new}$);
                $inserted \leftarrow true$;
                break;
        if $not\ inserted$ then
            Create new cluster $c_{new}$;
            $c_{new}$.insert_vertex($q_{new}$);
            $\mathcal{C}$.insert_cluster($c_{new}$);
```

The snapshot below presents the implementation of the Nearest Neighbour Search using clusters.

```
RrtConConBase::Neighbor YourPlanner::nearest(...)
{
    ...
    if (&tree == &this->tree[0])
    {
        for (auto& cluster : this->vertexClusterA)
        {
             // calculate distance to cluster's centroid
            ::rl::math::Real d = this->model->transformedDistance(
                    chosen, *tree[*cluster.first].q);
            if (d < minCentroidDistance)
            {
                minCentroidDistance = d;
                nearestCluster = cluster.second;
            }
        }
    }
    ...
    for (auto& vertex : *nearestCluster) {
        ... // check for closest node in nearest cluster
    }
    return p;
}
```

The following code snippet shows how new vertices are added to the cluster structure.

```
RrtConConBase::Vertex YourPlanner::addVertex(...)
{
    if (&tree == &this->tree[0])
    {
        // check if cluster structure is empty
        if (this->vertexClusterA.empty())
        {
            vertexClusterVector->push_back(clusterVertex);
            this->vertexClusterA[clusterVertex] =
                vertexClusterVector;
        } else {
            // remember if, vertex was added to a cluster
            bool inserted = false;
            // iterate through all clusters
            for (auto& cluster : this->vertexClusterA)
            {
                // calculate distance to cluster centroid
                ::rl::math::Real distance =
                    this->model->transformedDistance(
                    *tree[*cluster.first].q, *q);
                // check if distance is within threshold
                if (distance <= this->clusterDistanceThresh)
                {
                    // add vertex to the cluster and stop
                    vertexClusterVector = cluster.second;
                    vertexClusterVector->push_back(clusterVertex);
                    this->vertexClusterA[cluster.first] =
                        vertexClusterVector;
                    inserted = true;
                    break;
                }
            }
            if (!inserted) // if cluster does not match, create new one
            {
                vertexClusterVector->push_back(clusterVertex);
                this->vertexClusterA[clusterVertex] =
                    vertexClusterVector;
            }
        }
    }
    return addedVertex;
}
```

A.3 Extension 3: Exhausted Nodes

In order to circumvent the undesired behavior of the RRT algorithm occasioned by exhausted nodes, we implement an exhaustion counter for extension failures `exhaustCount` as a member variable of each vertex. Therefore, whenever a collision occurs during the expansion process, the counter is incremented by one, as shown in the snapshot of the following utilized code:

```
RrtConConBase::Vertex YourPlanner::connect(...)
{
    ...
    if (this->model->isColliding())
    {
        tree[nearest.first].exhaustCount++;
        break; // stop the extension in case of collision
    }
    ...
}
```

When a nearest neighbor is searched, the exhaustion counter starts to play its role. As primary check, the exhaustion counter of the to be checked node (vertex) is inspected. In case the value of it does not exceed the exhaustion threshold `exhaustedThresh`, which is *100* in our simulations, the node is used for the selection process.

```
RrtConConBase::Neighbor YourPlanner::nearest(...)
{
    ...
    if (tree[*vertex].exhaustCount <= this->exhaustedThresh)
    {
        p.first = *vertex;
        p.second = d;
    }
    ...
}
```

B Benchmark for the Extensions

The following table presents the results of the simulations. Each presented variation of the RRT is simulated ten times in order to be able to compute average values that are representative.

	No Ext.	No Ext. (rev.)	Ext. 1	Ext. 1 (rev.)
avgT [ms]	74006	56400	44211	50659
sdtT [ms]	39490	20790	14672	25963
avgNodes	11674	10905	7546	9902
avgQueries	584516	551001	420405	527119

	Ext. 1+2	Ext. 1+2 (rev.)	Ext. 1+2+3	Ext. 1+2+3 (rev.)
avgT [ms]	55947	42089	43587	48135
sdtT [ms]	26698	23615	20081	20956
avgNodes	11869	9199	13117	10889
avgQueries	712027	599322	782132	657502

Table 1 | Benchmark of simulations with different extensions.

Interpretation of the results

Table *Table 1* shows our benchmark results of our extensions. We conclude from this table, that **Extension 1** improves the runtime of the motion planning algorithm. This indicates that adjusting the sampling strategy based on the visible Voronoi regions helps to navigate the robot through narrow passages.

In combination with **Extension 2**, no performance improvement is notable, it even seems to decrease slightly. Several reasons for this seen behaviour can be given:

- The algorithm does not guarantee to find the nearest neighbor, instead it only finds an approximation.
- The worst time complexity is still $\mathcal{O}(n)$. This is the case, if each cluster only contains a single node or if all nodes are bundled in a single cluster. Both extrema can be balanced by the threshold. Decreasing the threshold leads to more clusters, containing less nodes. In contrast, increasing the threshold leads to less clusters that contain more nodes. Thus, a wrong choice of the threshold may result in a bad performance.
- The possible gain of exploiting the data structure to find a nearest neighbor faster does not compete with the additional computation and memory that is required to manage the clusters.

- The cluster may overlap, which may lead to some ambigiouty in the algorithm. Based on the current implementation, a node might not be added to the cluster with smallest distance to the centroid but to the first found cluster, where the node is within the hypersphere.

Adding **Extension 3** to the path finder, again increases the performance. Avoiding to extend nodes which failed too often when tried to be extended, leads to an extension of more promising nodes instead. Thereby the trees grow with a higher efficiency.

Starting the simulations with reverse order regarding the start and goal configuration (tree to expand), yields to significant different results. This can be contributed to the random seeding and performance inconsistency of the machines used for the simulations. Still, the combination of all three extensions performs the best.

C Implementation Table

The individual contributions of the group members are detailed in the table below (*Table 2*).

Student Name	Ext. 1 (impl.)	Ext. 1 (doc.)	Ext. 2 (impl.)	Ext. 2 (doc.)	Ext. 3 (impl.)	Ext. 3 (doc.)	Bench-mark
Jingsheng Lyu			X	X			
Prathamesh A. More				X			X
Lysander K. V. Rupp	X	X		X			X
Moritz D. P.-T. Vogt				X	X	X	X

Table 2 | Implementation table.

References

[1] A. Yershova, L. Jaillet, T. Simeon, and S. M. LaValle. Dynamic-domain rrts: Efficient exploration by controlling the sampling domain. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 3856–3861, 2005.

[2] L. Jaillet, A. Yershova, S. M. La Valle, and T. Simeon. Adaptive tuning of the sampling domain for dynamic-domain rrts. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2851–2856, 2005.

[3] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[4] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, USA, 2006.

[5] J.J. Kuffner and S.M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 2, pages 995–1001, San Francisco, CA, USA, 2000. IEEE.

[6] Kevin M Lynch and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge Univeristy Press, 2017.