SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER, CONTROL AND MANAGEMENT ENGINEERING

FP7
Improving Footstep Planning Algorithms by Efficient Nearest Neighbor Searching

AUTONOMOUS AND MOBILE ROBOTICS

Professor:
Giuseppe Oriolo

Supervisor:
Michele Cipriano

Students:
Ionut Marian Motoi
Leonardo Saraceni
Giuseppe Sensolini Arrà

Academic Year 2020/2021

Contents

1 Introduction

One of the bottlenecks in the performance of sampling-based motion planning algorithms is the computational cost of the nearest-neighbor operation. Consequently the development of efficient techniques for nearest neighbor searching is of paramount importance, in order to keep the planning time low.

As stated by Michal Kleinbort et al. in the paper [8], the complexity of nearest-neighbor search dominates the asymptotic running time of many sampling-based motion-planning algorithms. In spite of this, collision detection is often considered to be the computational bottleneck in practice. However, there are many asymptotically optimal planning algorithms, that are called *NN-sensitive*, in which the practical computational role of the nearest-neighbor search is far from being negligible, even for a relatively small number of samples. This means that the portion of running time taken up by nearest-neighbor search is comparable to, or sometimes even greater than, the portion of time taken up by collision detection. This reinforces and substantiates the claim that motion-planning algorithms could **significantly benefit** from efficient nearest-neighbor data structures.

In order to show this, in the figure below (fig. 1) we can see that the total time spent by the algorithm is completely dominated by the search time.

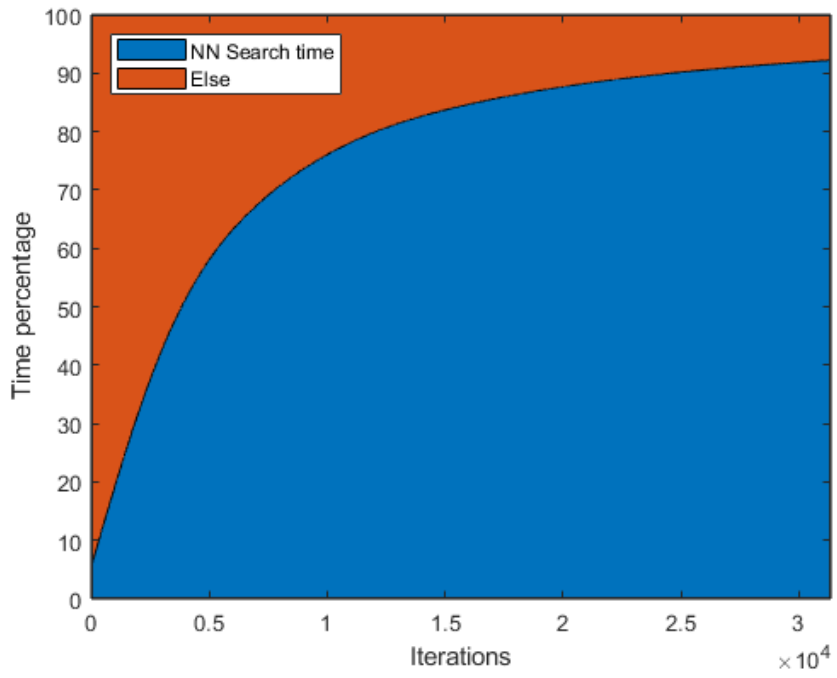

Figure 1: Time spent doing NN search vs RRT total time using Naive algorithm

Therefore, the aim of the project is to enhance the footstep planning of humanoid robots, improving the nearest neighbor searching algorithm, by using a different data structure, such as the k-d tree, that allows to perform searches in logarithmic time. More specifically our goal consists in extending the RRT and RRT* footstep planners proposed by our tutor, by implementing a self-balancing k-d tree on non-Euclidean topologies.

In order to validate the new method, simulations have been performed using V-REP/CoppeliaSim, using the humanoid robot HRP4 framework for the tests.

2 Sampling-Based Planning

The motion planning problem consists of finding a dynamically feasible trajectory that takes the robot from an initial state to a goal state while avoiding collision with obstacles.
This is of fundamental importance especially for Humanoid robots, that must be able to navigate autonomously, even in complex and cluttered environments. For this to be possible, they must be able to quickly and efficiently compute desired footsteps to reach a goal.

Many approaches have been considered in order to obtain always increasing performances to solve the footstep planning problem. Rapidly-exploring random trees (RRT) is a common option that both creates a graph and finds a path. The path will not necessarily be optimal. RRT*, is an optimized modified algorithm that aims to achieve a shortest path, whether by distance or other metrics.

There are two main challenges that are imposed by this problem. First, the robot does not have existing nodes to travel between. Secondly, one must determine how a shortest path will be determined. A possible way to solve footstep planning problems is by using continuous optimization techniques, which do not restrict possible steps to a finite set. However, this would require an expensive computation in order to find the obstacle-free regions. The alternative, is to use only a finite set of predefined steps for fast collision checking, and therefore the solution consists of a particular sequence of such movements.

2.1 RRT

The Rapidly-exploring Random Tree (RRT) is a classical algorithm of motion planning based on incremental sampling, which is widely used to solve the planning problem of mobile robots, since it is able to efficiently search non-convex, high-dimensional spaces by randomly building a space-filling tree. In particular, RRT (Alg. 1) consists of a tree of paths incrementally grown at each iteration, by selecting a random configuration from the search space. It starts with an empty tree and adds a node that corresponds to the initial state. The for N iterations it expands the tree by sampling a random state z_{rand} from the obstacle-free space and finds the closest node (with respect to a predefined metric) in the tree $z_{nearest}$. Then it computes the trajectory x_{new} that expands the closest node toward the sample. After that, if the trajectory is feasible, the new state z_{new} is added to the tree with $z_{nearest}$ as its parent and continues with the next iteration.

It is important to notice that the the expansion is biased towards unexplored areas of the search space, that in this way can be rapidly covered. RRT can easily handle problems with obstacles and differential constraints, and for this

reason has been widely used in autonomous robotic motion planning.

Algorithm 1 $\mathcal{T} = (V, E) \leftarrow RRT(z_{init})$

1: $\mathcal{T} \leftarrow InitializeTree()$
2: $\mathcal{T} \leftarrow InsertNode(\emptyset, z_{init}, \mathcal{T})$
3: **for** $i = 1$ *to* $i = N$ **do**
4: $z_{rand} \leftarrow Sample(i)$;
5: $z_{nearest} \leftarrow Nearest(\mathcal{T}, z_{rand})$
6: $(x_{new}, u_{new}, \mathcal{T}_{new}) \leftarrow Steer(z_{nearest}, z_{rand})$
7: **if** $ObstacleFree(x_{new})$ **then**
8: $\mathcal{T} \leftarrow InsertNode(z_{nearest}, z_{new}, \mathcal{T})$
9: **return** T

The benefit of the algorithm is its speed and ease of implementation. Compared to other path planning algorithms, RRT is fairly quick. The most expensive part of the algorithm is finding its closest neighbor, since the cost of this process depends heavily on the number of vertices that have been generated.

2.2 RRT*

An alternative algorithm proposed by Karaman and Frazzoli [7] is RRT*, a sampling-based method that has the asymptotic optimality property along with probabilistic completeness guarantees. Like the RRT, it quickly finds a feasible motion plan. Then, it improves the current plan toward the optimal solution in the remaining time before the plan execution is complete.

The RRT* algorithm (Alg. 2) solves the optimal motion planning problem by building and maintaining a tree T made of vertices representing states in X_{free}. The tree is generated in a similar way as the standard RRT, with the addition of the optimization part.

The RRT* starts with an empty tree and adds a single node corresponding to the initial state. Then it builds and refines the tree incrementally through a set of N iterations (lines 3-11). At each of those iterations it samples a random state z_{rand} from the obstacle-free space (line 4) and solves for a trajectory x_{new} that extends the closest node in the tree $z_{nearest}$ toward the sample (lines 5-6). If this is a feasible trajectory, i.e. does not collide with obstacles (line 7), RRT* considers all the nodes in the neighborhood of z_{new} (line 8) and evaluates the cost of choosing each of them as the parent.

This procedure (Alg. 3) computes the total cost as the additive combination of the cost associated with reaching the potential parent node and the cost of the trajectory to z_{new}. Then the node with the lowest cost z_{min} becomes the parent and the new node z_{new} is added to the tree (line 10 - Alg. 2).

Finally the ReWire process (Alg. 4) checks each node z_{near} in the vicinity

Algorithm 2 $\mathcal{T} = (V, E) \leftarrow RRT^*(z_{init})$

$\mathcal{T} \leftarrow InitializeTree()$
$\mathcal{T} \leftarrow InsertNode(\emptyset, z_{init}, \mathcal{T})$
for $i = 1$ *to* $i = N$ **do**
    $z_{rand} \leftarrow Sample(i)$
    $z_{nearest} \leftarrow Nearest(\mathcal{T}, z_{rand})$
    $(x_{new}, u_{new}, \mathcal{T}_{new}) \leftarrow Steer(z_{nearest}, z_{rand})$
    if $ObstacleFree(x_{new})$ **then**
        $Z_{near} \leftarrow Near(\mathcal{T}, z_{new}, |V|)$
        $z_{min} \leftarrow ChooseParent(Z_{near}, z_{nearest}, z_{new}, x_{new})$
        $\mathcal{T} \leftarrow InsertNode(z_{min}, z_{new}, \mathcal{T})$
        $\mathcal{T} \leftarrow Rewire(\mathcal{T}, Z_{near}, z_{min}, z_{new})$
return T

Algorithm 3 $z_{min} \leftarrow ChooseParent(Z_{near}, z_{nearest}, x_{new})$

$z_{min} \leftarrow z_{nearest}$
$c_{min} \leftarrow Cost(z_{nearest}) + c(x_{new})$
for $z_{near} \in Z_{near}$ **do**
    $(x', u', \mathcal{T}') \leftarrow Steer(z_{near}, z_{new})$
    if $ObstacleFree(x')$ **and** $x'(\mathcal{T}') = z_{new}$ **then**
        $c' = Cost(z_{near}) + c(x')$
        if $c' < Cost(z_{new})$ **and** $c' < c_{min}$ **then**
            $z_{min} \leftarrow z_{near}$
            $c_{min} \leftarrow c'$
return z_{min}

Algorithm 4 $\mathcal{T} \leftarrow ReWire(\mathcal{T}, Z_{near}, z_{min}, z_{new})$

for $z_{near} \in Z_{near} \setminus \{z_{min}\}$ **do**
    $(x', u', \mathcal{T}') \leftarrow Steer(z_{new}, z_{near})$
    if $ObstacleFree(x')$ **and** $x'(\mathcal{T}') = z_{near}$
    and $Cost(z_{new}) + c(x') < Cost(z_{near})$ **then**
        $\mathcal{T} \leftarrow ReConnect(z_{new}, z_{near}, \mathcal{T})$
return $\mathcal{T}$

of z_{new} to see if reaching z_{near} via z_{new} would achieve a lower cost than doing so through its current parent (line 3). When this connection reduces the total cost associated with z_{near}, the algorithm "rewires" the tree to make z_{new} the parent of z_{near} (line 4).

The RRT* then continues with the next iteration.

2.3 Anytime

In practice robotic motion planning algorithms must operate with limited computational resources, as well as with incomplete and often imperfect knowledge of the environment in which it navigates. In such cases, anytime algorithms are well suited since they quickly find a feasible, but not necessarily optimal motion plan, then incrementally improve it over time toward optimality.

A system that uses anytime planning needs to perform two functions: *execution* and *optimization* of the current plan (alg. 5).

The properties such an algorithm need to have are some form of completeness guarantee (sampling-based algorithms are usually probabilistically complete) and asymptotic optimality.

RRT has the first property. In fact it is able to efficiently find a feasible solution. However, it lacks the asymptotic optimality property. Karaman and Frazzoli proved that the probability of the RRT algorithm converging to an optimal solution is actually zero [7]. In any case, it is possible to use RRT with an anytime modality, by first computing a feasible solution and then using the remaining time to improve the initial plan, even if it will probably never be the optimal one.

On the other hand, RRT* has both those properties and for this reason it is preferable, especially in real time motion planning applications.

Algorithm 5 Anytime

1: $\mathcal{T} \leftarrow RRT*(v_{init})$
2: **repeat**
3: $\quad \mathcal{P} \leftarrow ExtractBestPlan(\mathcal{T})$
4: $\quad v_{curr} \leftarrow \mathcal{P}_0$
5: $\quad v_{next} \leftarrow \mathcal{P}_1$
6: $\quad$ **do in parallel**
7: $\quad\quad ExecuteStep(v_{curr}, v_{next})$
8: $\quad\quad \mathcal{T} \leftarrow RRT^*(v_{next})$
9: **until** v_{next} *in* $\mathcal{G}$

2.4 Footstep Planning

To increase the speed of operation and reduce operator burden, humanoid robots must be able to function autonomously, even in complex, cluttered environments. For this to be possible, they must be able to quickly and efficiently compute desired footsteps to reach a goal.

While the mobility of humanoid robots has significantly improved, allowing them to walk quickly and robustly, they are still fairly limited as to their ability to handle rough terrain. Rough terrain for robots has a sparse, limited number of footholds, with large, discrete height changes and varied surface normals. This requires the robot to use accurate foot placement in order to reach the goal. Navigating over rough terrain is a key skill for humanoid robots to function in many operating environments, and highlights their distinctive mobility capabilities.

In order to achieve the *walk to* task of the robot, an offline planner is provided, whose target is to find a proper sequence of footsteps $\mathcal{P} = f^j$ that brings the robot to the goal region $\mathcal{G}$, together with the associated swing foot trajectories f^j_{swg}.

Our project is based on the work related to the paper [1] by Ferrari et al., where a humanoid robot has to reach an assigned goal region $\mathcal{G}$ (*walk-to* task) in an environment (World of Stairs) made of horizontal patches located at different heights. The environment is represented as a 2.5-dimensional grid map of equally-sized cells, also called *elevation map*, denoted by $\mathcal{M}_z$, used to provide the height of the ground at each cell of coordinate *x, y*.

In particular, the footstep planner consists of a randomized algorithm, that iteratively builds a tree in the search space. Each vertex specifies the poses of both feet during a double support phase, and then, during the walk phase, one of the two feet will be chosen as *support*, while the other is the moving one (*swinging*). The initial support foot can be chosen arbitrarily. At each iteration, a random point p_{rand} is randomly generated by drawing its *x, y* coordinates, and then the closest vertex in the tree is taken (according to the chosen metric), together with the foot poses associated to it. It's important to notice that the z coordinate is simply retrieved from the *elevation map* $\mathcal{M}_z$. After this, a candidate footstep f_{cand} is generated by randomly selecting a final pose of the swinging foot from the catalogue of primitives (fig. 2), defined with respect to the support foot of the closest vertex. At this point, the algorithm verifies if the candidate is feasible and and if a collision-free trajectory exists, such that the swinging foot can be moved to the candidate pose, and the new vertex is added to the tree, connected to its parent.

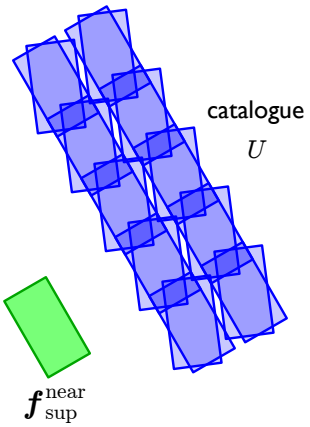

Figure 2: Catalogue of primitives

Algorithm 6 *FootstepPlanner*

root the tree $\mathcal{T}$ at $v_{ini} \leftarrow (f_L, f_R)$
$i \leftarrow 0$
repeat
    $i \leftarrow i + 1$
    generate a random point p_{rand} on the ground
    select the closest vertex v_{near} in $\mathcal{T}$ to p_{rand} according to $\gamma(\cdot, p_{rand})$
    randomly select a candidate footstep f^{cand} from primitive catalogue U
    if f^{cand} is feasible **then**
        $h \leftarrow h_{min}$
        $p^{cand}_{swg} \leftarrow BuildTrajectory(f^{near}_{swg}, f^{cand}, h)$
        while $h \leq h_{max}$ **and** $Collision(p^{cand}_{swg})$ **do**
            $h \leftarrow h + \Delta h$
            $p^{cand}_{swg} \leftarrow BuildTrajectory(f^{near}_{swg}, f^{cand}, h)$
        if $h \leq h_{max}$ **then**
            $v_{new} \leftarrow (f^{cand}, f^{cand}_{sup})$
            add vertex v_{new} to $\mathcal{T}$ as a child of v_{near}
            compute midpoint **m** between the feet at v_{new}
until $m \in \mathcal{G}$ **or** $i = i_{max}$

3 Nearest Neighbor

One of the bottlenecks in the performance of sampling-based motion-planning algorithms is the cost of nearest-neighbors calls. Hence, it is necessary to develop efficient techniques that allow searching for the nearest-neighbor in configuration spaces that are used in motion planning. Our implementation of the nearest-neighbor search algorithm is based on the one developed by Lavalle et. al. that uses kd-trees.

Nearest neighbor search (**NNS**), is the optimization problem of finding the point in a given set that is closest (or most similar) to a given point. Formally, the nearest-neighbor search problem is defined as follows: given a set S of points in a space $\mathcal{X}$ and a query point $q \in \mathcal{X}$, find the closest point in S to q.
Closeness is typically expressed in terms of a dissimilarity function: the less similar the objects, the larger the function values. In general, many different dissimilarity functions can be used, like the Euclidean distance, Manhattan distance or other distance metrics. However, the dissimilarity function can be arbitrary. In fact, in our case we had to find the double support configuration v whose support foot is the closest to a 3-dimensional query point Pt at each iteration, using a particular metric, given by:

$$\gamma(v, Pt) \leftarrow D(m, Pt) + \alpha|\theta_p| \tag{1}$$

Where α is a positive scalar, and $|\theta_p|$ is the angle between the robot sagittal axis and the line joining the double support configuration v to the 3-dimensional point Pt.
Various solutions to the NNS problem have been proposed. The quality and usefulness of the algorithms are determined by the time complexity of queries as well as the space complexity of any search data structure that must be maintained. In particular our task was to implement a **balanced** k-d tree in order to obtain a remarkable improvement during the search phase.

3.1 Naive algorithm

The simplest solution to the nearest neighbor search problem is to compute the distance from the query point to every other point in the search space, keeping track of the best one. This algorithm, sometimes referred to as the naive approach, has a running time of O(n), where n is the number of nodes in the searching tree. There are no search data structures to maintain, so linear search has no space complexity beyond the storage of the tree.
The previous implementation of the search was linear in time, in fact it consisted only of a simple Depth First Search (DFS) on the current tree, and

its implementation is reported here briefly.

Algorithm 7 *LinearSearch*

1: **for** $v \in \mathcal{T}$ **do**
2: $d \leftarrow \gamma(v, Pt)$
3: **if** $d < Curr_{Min}$ **then**
4: $Curr_{Min} = d$
5: $Nearest_{Node} = v$
6: **return** $Nearest_{Node}$

However this solution resulted to be quite slow, therefore the key of our project is to speed up this process, in order to make its time complexity logarithmic in time.

3.2 Static k-d tree

A k-d tree is a space-partitioning data structure for organizing points in a k-dimensional space. K-d trees are a useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches) and creating point clouds.
A k-d tree is built by mean of two kind of nodes.

- **Leaf Node**: contains a k-dimensional point.
- **Middle Node**: represents a splitting hyperplane dividing the space in two parts. Points to the left of this hyperplane are represented by the left subtree of that node and points to the right of the hyperplane are represented by the right subtree. In our implementation this kind of node does not contain any k-dimensional point.

The proposed k-d tree make use of hyperplanes parallel to the main axis. The algorithm iteratively changes the cutting dimension and at each step computes the cutting value along the current cutting dimension. The construction of the k-d tree is explained in detail the following example.

example. Consider a set of points $\{a, b, c, d\} \in \mathbb{R}^2$, and let the *cutting dimension* be x. Then the set is stored in ascending order (according to the x coordinate) in a vector $\mathtt{V} = \{b, a, c, d\}$. The first $\lfloor \frac{|\mathtt{V}|}{2} \rfloor$ points of $\mathtt{V}$ are stored in a new vector $\mathtt{V}_L$, while the remaining ones are placed in $\mathtt{V}_R$; In this particular example $\mathtt{V}_L = \{b, a\}$ and $\mathtt{V}_R = \{c, d\}$. Ultimately the *cutting value* is computed as $(a_x + c_x)/2$, and the cutting dimension is switched to y. The same steps are recursively applied to $\mathtt{V}_L$ and $\mathtt{V}_R$ until they contains 1 point only.

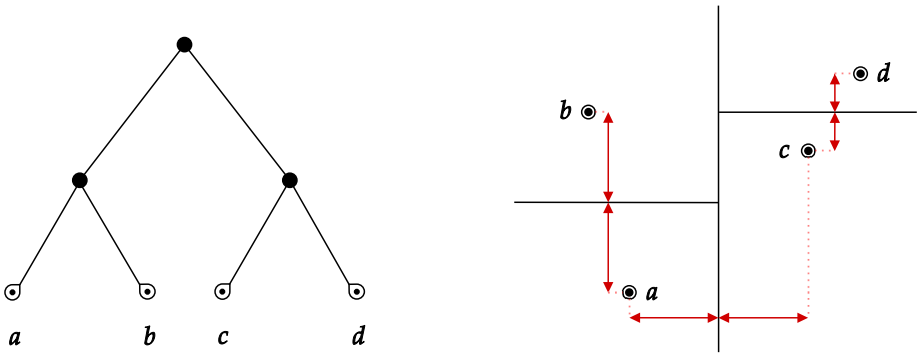

Figure 3: example of k-d tree containing 4 elements

Algorithm 8 $BuildKDTree(V, cut\_dim)$

1: **if** $|V| = 1$ **then**
2: **return** $LeafNode(V)$
3: split V along cut_dim in V_L and V_R
4: update cut_dim
5: $T_L \leftarrow BuildKDTree(V_L, cut\_dim)$
6: $T_R \leftarrow BuildKDTree(V_R, cut\_dim)$
7: **return** $MiddleNode(cut\_dim, cut\_val, T_L, T_R)$

The splitting in line 4 of algorithm 8 is done in a way that guarantees the balancing of the constructed k-d tree. Line 5 swap the cutting dimension from x to y and viceversa.
Nearest neighbor search over the k-d tree is accomplished in $O(\log n)$. The *search* is summarized in algorithm 9. In what follows d_{box} is the squared distance from the root box to the *query*, b_K is the projection of the current node onto the space stored in the node, and p the closest point to the query; d_{best} and p variables are initialized respectively to ∞ and $NULL$.

Algorithm 9 $Search(query, d_{box}, d_{best}, p)$

1: **if** $CurrentNode$ is a $MiddleNode$ **then**
2: **if** $d_{box} < d_{best}$ **then**
3: Split b_K into subboxes b_{K_L} and b_{K_R} corresponding to T_L and T_R
4: $d_L \leftarrow dist^2(query, b_{K_L})$
5: $d_R \leftarrow dist^2(query, b_{K_R})$
6: **if** $d_L < d_R$ **then**
7: $T_L \rightarrow Search(d_{box}, d_{best}, p)$
8: $T_R \rightarrow Search(d_{box} - d_L + d_R, d_{best}, p)$
9: **else**
10: $T_R \rightarrow Search(d_{box}, d_{best}, p)$
11: $T_L \rightarrow Search(d_{box} - d_R + d_L, d_{best}, p)$
12: **if** $CurrentNode$ is a $LeafNode$ **then**
13: Compute the squared distance from the *query* to the leaf element
14: Update p and d_{best}

3.3 Making the kd-tree dynamic

The main goal of this project is the implementation of a different data structure allowing us to perform the search in $O(\log^2 n)$ instead of $O(n)$. In order to achieve this result we have been inspired by the **Logarithmic method** exposed in [4]. This method allows to perform a *preprocess* of the input set creating a data structure on it. By this, it is possible to answer any query with much lower cost than if we did not have the structure, more precisely in logarithmic time.

To achieve such result, we need to make our structure dynamic. Namely, we want to support the *insertion* of elements of the search space in the binary structure. This method assume that we have already designed a static structure. Let the input set be initially empty, then we want to maintain a *semi-dynamic structure* as new elements are inserted into the structure. Note that in this context we refer to a semi-dynamic as to a structure over which the *insertion* update is allowed, while the *deletion* update is not defined; this simplification is possible because the dataset of our application only expands, but never shrinks. Basically, the logarithmic method can convert a static structure to a semi-dynamic structure. It consumes $O(n)$ space, answers a query in $O(\log^2 n)$ time, and supports an insertion in $O(\log^2 n)$ **amortized time**.

Let n=$|S|$ be the number of elements that have been inserted so far, then at each iteration, S is divided into $h = \lfloor \log_2 n \rfloor + 1$ mutually disjoint partitions $S_0, ..., S_{h-1}$. Partition S_i either is empty or must have size 2^i $\forall i \in [0, h-1]$.

Let us call e the new element to be added to S. We first find i^*, i.e., the smallest $i \ni S_i = \emptyset$. Then

- If $i^* = 0$ the set S_0 is created with only e itself.
- If $i^* > 0$ a new set $S_{i^*} = e \oplus S_0 \oplus S_1 \oplus ... \oplus S_{i^*-1}$ is defined; then $S_0, ..., S_{i^*-1}$ are cleared and the related k-d trees destroyed, while a new k-d tree is built from scratch with the elements of S_{i^*}.

example 1. Consider the case in which the structure S contains 5 elements, therefore the number of partitions is $h = \lfloor \log_2 5 \rfloor + 1 = 3$. Elements are stored into S_0, S_1 and S_2 according to the binary representation of n; the full structure S is represented in figure 4.

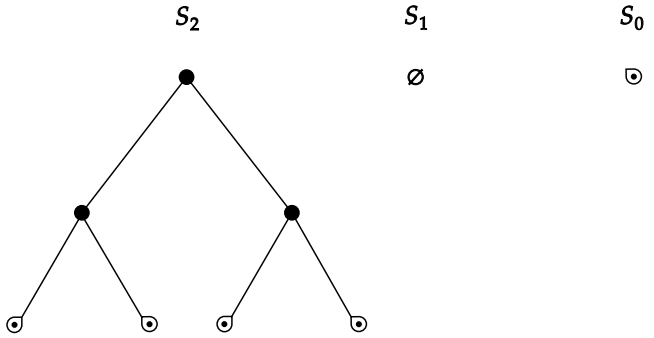

Figure 4: Structure with 5 elements

Suppose now that a new element e needs to be *inserted* in S; each bit of `101` (the binary representation of n) is evaluated from right to left.
The first bit encountered is a 1: the element contained in S_0 is stored and the k-d tree deleted.
The second bit is 0: a k-d tree S_1 is created. it contains the e and the element previously contained in S_0.
The obtained structure is represented in figure 5. Note that the k-d tree S_2 is not affected by this operation.

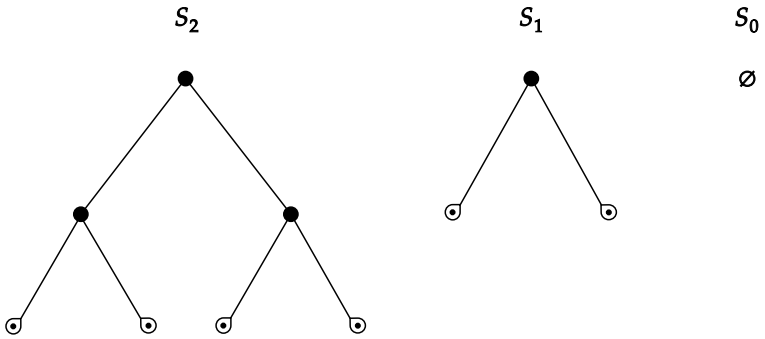

Figure 5: Structure with 6 elements

example 2. Consider now the situation in which 7 elements are already stored in S, and a new one (e) needs to be inserted. In this case the starting binary representation is 111. This implies that all ktrees are destroyed and and all the elements are stored in a new k-d tree S_3 along with the new element e. The final result is depicted in figure 6.

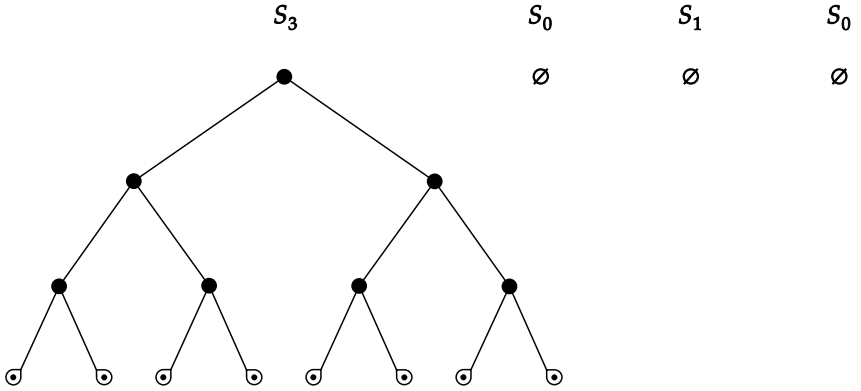

Figure 6: Structure with 8 elements

The insertion procedure is summarized in algorithm 10.

Algorithm 10 *Insertion*(e)

```
1: initialize vector V
2: add e to V
3: for i ∈ [0, h − 1] do
4:     if S_i==∅ then
5:         S_i ← BuildKDTree(V)
6:         break
7:     else
8:         add elements of S_i to V
9:         S_i ← ∅
```

4 Simulations

In order to verify our results, we performed various simulations using the HRP4 humanoid robot framework in the CoppeliaSim environment. We compared the performance of the previous framework with the new framework that uses K-D Trees for the Nearest Neighbor search. The test were performed in two different scenarios, both in which the robot had to reach a goal region. Since the footstep planner is randomized, to obtain a better comparison of the results, we performed a set of 100 simulations for each combination of algorithm, planner, scenario and time budget.

All the simulations have been performed on an Intel(R) Core(TM) i5-6200U CPU running at 2.30 GHz (2 cores, 4 threads).

In the first scenario, named Corridor (fig. 7), in order to reach the goal region the robot has to pass through a narrow corridor and then through a room that has a little height variation with respect to the rest of the environment.

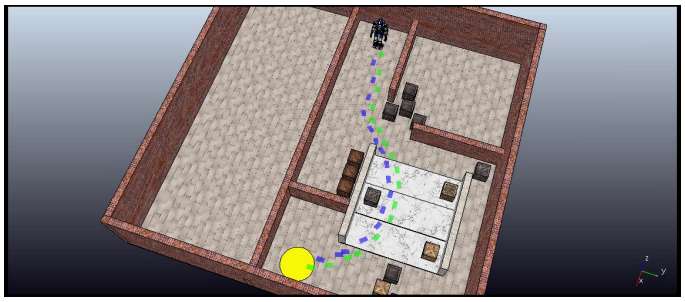

Figure 7: Corridor scenario

The second scenario, named Ditch (fig. 8), is a little more challenging with respect to the above, since there are more height variations, of different entity. Therefore, this represents an even greater challenge, since the robot cannot execute steps of any height. This fact is clearly visible, since the path the robot chooses doesn't goes straight through the middle of the platform, but has to pass through an intermediate platform on the top side of the map.

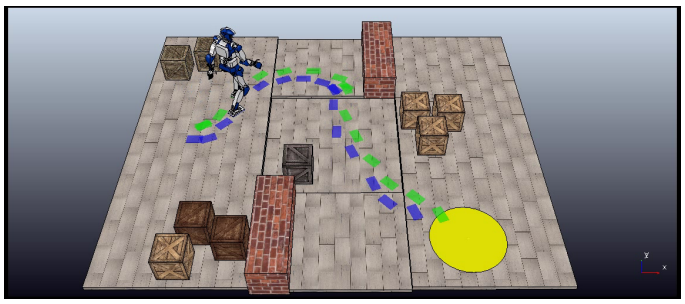

Figure 8: Ditch scenario

Tables 1 - 4 report some data obtained by averaging the results of the 100 simulations on each scenario with both the algorithm using the naive approach to the nearest neighbor problem and with our algorithm using KD-Tree. We tested those algorithms with RRT and RRT* planners with 1s, 5s, 10s and 25s of time budget and also by stopping the planner as soon as a goal is found with a max time budget of 100s.

Algorithm	ΔT [s]	Avg Cost	Min Cost	Max Cost	Iters	Tree Size	Successes
Naive	avg: 1.786	67.530	48	102	7819.2	2044.9	100/100
	1.0	61.933	42	83	5353.0	1657.9	30/100
	5.0	68.147	47	95	15489.1	3160.4	95/100
	10.0	66.293	48	101	21690.7	4785.9	99/100
	25.0	67.610	46	100	37912.1	7427.4	100/100
KDTree	avg: 0.679	70.420	48	99	8592.2	2302.6	100/100
	1.0	66.822	51	91	12396.6	3213.4	90/100
	5.0	66.920	49	97	52497.2	10021.1	100/100
	10.0	66.940	49	93	94667.8	16135.6	100/100
	25.0	67.990	41	94	198823.4	30154.9	100/100

Table 1: Planner:RRT - Environment:Corridor

Algorithm	ΔT [s]	Avg Cost	Min Cost	Max Cost	Iters	Tree Size	Successes
Naive	avg: 0.994	49.950	33	70	4855.8	1676.2	100/100
	1.0	49.224	32	67	5518.1	1915.6	67/100
	5.0	49.646	28	73	12332.1	3971.6	99/100
	10.0	49.520	33	68	16501.8	5129.9	100/100
	25.0	48.830	36	83	28750.6	8379.3	100/100
KDTree	avg: 0.429	50.430	32	71	4895.5	1714.4	100/100
	1.0	51.222	32	75	10029.7	3364.6	99/100
	5.0	49.770	34	64	35470.0	10344.6	100/100
	10.0	49.110	31	74	52283.4	14658.4	100/100
	25.0	48.830	30	65	124247.8	31255.7	100/100

Table 2: Planner:RRT - Environment:Ditch

The results show that there is an increment in performance with the new algorithm for each combination of planner and time budget. This is because with an improved nearest neighbor search, the planners are able to expand more the tree and perform more iterations in the same amount of time, resulting in a greater rate of success for both RRT and RRT* and in a lower average cost of the footstep plan in the case of RRT*.

Algorithm	ΔT [s]	Avg Cost	Min Cost	Max Cost	Iters	Tree Size	Successes
Naive	avg: 3.532	53.320	40	75	8886.2	2016.1	100/100
	1.0	51.200	43	59	4084.2	1266.8	10/100
	5.0	47.554	37	62	12041.7	2750.9	83/100
	10.0	46.333	38	58	18056.9	3768.7	96/100
	25.0	42.140	33	60	31126.4	5664.4	100/100
KDTree	avg: 2.467	52.210	42	66	8432.9	2149.9	100/100
	1.0	48.350	32	56	5055.4	1546.8	20/100
	5.0	46.913	36	62	16818.2	3886.8	93/100
	10.0	42.619	34	55	25650.6	5386.4	97/100
	25.0	38.390	32	53	53499.1	9670.2	100/100

Table 3: Planner: RRT* - Environment: Corridor

Algorithm	ΔT [s]	Avg Cost	Min Cost	Max Cost	Iters	Tree Size	Successes
Naive	avg: 1.479	35.370	23	46	4938.2	1580.3	100/100
	1.0	33.571	26	43	3851.1	1336.1	28/100
	5.0	30.857	21	39	10376.2	3257.7	98/100
	10.0	29.820	21	39	13654.8	4109.6	100/100
	25.0	26.880	20	33	20044.4	5545.3	100/100
KDTree	avg: 1.002	35.630	25	46	4817.2	1653.6	100/100
	1.0	33.864	25	45	4932.5	1726.5	44/100
	5.0	28.270	20	38	14740.3	4616.4	100/100
	10.0	26.650	19	33	22652.3	6767.9	100/100
	25.0	24.010	18	35	39012.9	10325.3	100/100

Table 4: Planner: RRT* - Environment: Ditch

We also tested the RRT* planner in anytime modality with a time budget of 1s in both scenarios. The results are summarized in Table 5. Also in this case, our algorithm was able to outperform the previous implementation. With the same time budget it is able to expand the tree by more than double obtaining a perfect rate of success while also having a smaller average cost.

Algorithm	Environment	Avg Cost	Min Cost	Max Cost	Iters	Tree Size	Successes
Naive	Corridor	41.090	32	48	2672.6	1392.9	100/100
	Ditch	27.730	21	34	2287.7	1157.9	100/100
KDTree	Corridor	38.730	32	50	3673.9	1762.7	100/100
	Ditch	26.400	18	32	3254.6	1444.9	100/100

Table 5: Planner: RRT* Anytime - Time Budget: 1s

5 Conclusions

We have successfully implemented an algorithm for performing an efficient nearest neighbor search in the context of path panning for humanoid robots. The proposed solution make use of the logarithmic method, which is based on k-dimensional trees stored in a semi-dynamic structure. Simulations in section 4 reveal satisfactory results, with both RRT and RRT* outperforming the previous work in almost every metric.

The optimization of the nearest neighbor search becomes more and more important as the paths to be planned become longer. This is because the the time spent doing nearest neighbor search in sampling-based planners approaches 100% of the total time used by the planner as time approaches infinity. In the figures below, we report the fraction of time spent performing the nearest neighbor search with respect to the total time used by the RRT planner as iterations grow.

In the figure [9], we can see that the Naive algorithm, uses a great amount of time in during the search phase. In particular it grows very rapidly and reaches more than 90% of the total time, which is not negligible.

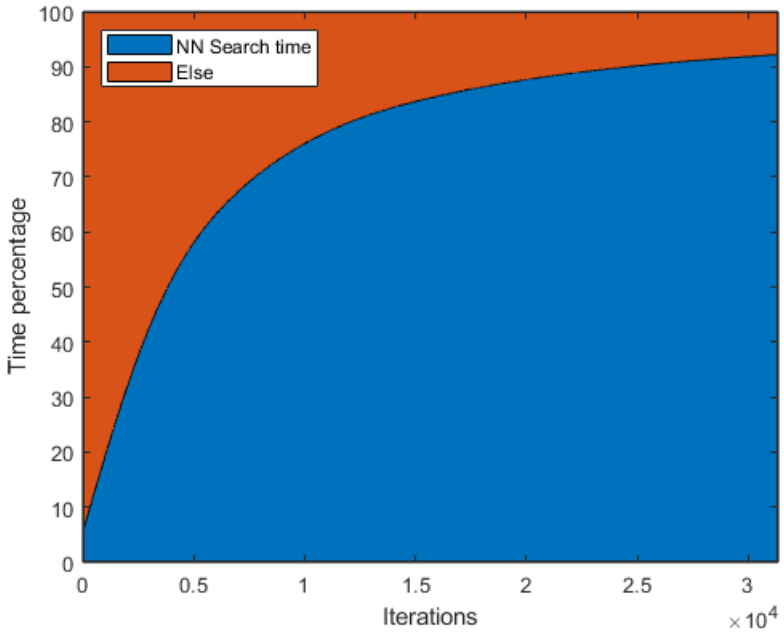

Figure 9: Time spent doing NN search vs RRT total time using Naive algorithm

On the other hand, in figure [10] are reported the K-D tree results, and we can notice that the time spent for the search, grows much slower than it was doing in the previous case. In fact, it barely reaches 60% of the total time after 5

times the iterations of the naive case. This results in a remarkable improvement, that was exactly the aim of this project.

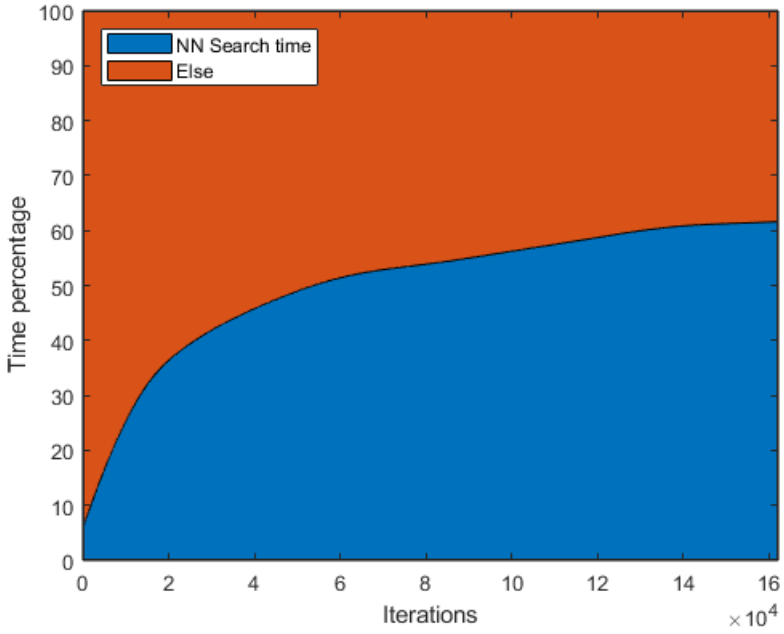

Figure 10: Time spent doing NN search vs RRT total time using K-D Tree algorithm

Possible future improvements could be to further optimize the dynamic k-d tree by avoiding too many rebuilds of the trees. Another optimization would be implementing a *delete* method that would make the k-d tree fully dynamic, allowing for more advanced techniques such as pruning the tree whenever it becomes too big. Finally, another improvement could be allowing the leaves to contain more than one data point.

References

[1] Paolo Ferrari, Nicola Scianca, Leonardo Lanari, Giuseppe Oriolo, *An Integrated Motion Planner/Controller for Humanoid Robots on Uneven Ground*, ECC 2019.

[2] Anna Yershova, Steven M. LaValle, *Improving Motion-Planning Algorithms by Efficient Nearest-Neighbor Searching*, IEEE Transaction on Robotics, vol. 23, no. 1, February 2007.

[3] Jon Louis Bentley, James B. Saxe, *Decomposable Searching Problems: Static-to-Dynamic Transformation*, Department of Computer Science, Carnegie-Mellon University, 1980.

[4] Yufei Tao, *Lecture Notes: The logarithmic method*, Chinese University of Hong Kong, 2012.

[5] Sunil Arya, David M. Mount, *Algorithms for Fast Vector Quantization*, Proc. IEEE Data Compress. Conf., Mar. 1993, pp.381-390.

[6] Sertac Karaman, Matthew R. Walter, Alejandro Perez, Emilio Frazzoli, *Anytime Motion Planning using the RRT**, IEEE International Conference on Robotics and Automation, ICRA 2011.

[7] Sertac Karaman, Emilio Frazzoli, *Incremental sampling-based algorithms for optimal motion planning*, Proc. Robotics: Science and Systems (RSS), 2010.

[8] Michal Kleinbort, Oren Salzman, Dan Halperin *Collision detection or nearest-neighbor search? On the computational bottleneck in sampling-based motion planning*, Algorithmic Foundations of Robotics XII: Proceedings of the Twelfth Workshop on the Algorithmic Foundations of Robotics, pag. 624–639, 2016.