Deformable Rapidly-Exploring Random Trees

Florian Hauer  
School of Aerospace Engineering  
Georgia Institute of Technology, Atlanta, GA 30332  
Email: fhauer3@gatech.edu

Panagiotis Tsiotras  
School of Aerospace Engineering  
Institute for Robotics and Intelligent Machines  
Georgia Institute of Technology, Atlanta, GA 30332  
Email: tsiotras@gatech.edu

Abstract—In this paper, using the Hypercube Diagonal Experiment we first investigate the convergence rates of sampling-based path-planning algorithms in terms of the dimensionality of the search space. We show that the probability of sampling a point that improves the solution decreases exponentially with the dimension of the problem. We then analyze how the samples can be repositioned in the search space in order to minimize the approximation error. Finally, we present the DRRT (Deformable Rapidly Exploring Random Tree) algorithm that utilizes optimization of sample location in the framework of RRT algorithms to improve convergence. It is shown that the DRRT algorithm significantly outperforms all current sampling-based algorithms in terms of convergence.

I. INTRODUCTION

The problem of path-planning in a $d$-dimensional continuous search space $S$ is considered in this paper. Specifically, the problem is to find a feasible path $\pi = [\pi_0, \pi_1, \ldots, \pi_N]$ for some $N > 0$, starting from $\pi_0 = x_{\text{start}} \in S$ to $\pi_N = x_{\text{goal}} \in S$, that minimizes a cumulative cost function of the form

$$J = \sum_{i=1}^{N} c(\pi_{i-1}, \pi_i),$$

where $c(\pi_{i-1}, \pi_i)$ is the cost to go from $\pi_{i-1}$ to $\pi_i$. We assume in this paper that the cost function $c$ is differentiable.

This problem is well-studied in the literature, and the family of Rapidly-Exploring Random Trees (RRT) algorithms is a very popular method to find a solution. All the algorithms in the RRT family share the same algorithmic structure, that is, they build a policy, represented as a tree $T$, to go from $x_{\text{start}}$ to any other point in the space $S$, based on a set of iteratively sampled points of $S$.

The original RRT algorithm [9] simply connects the new sample to the closest point in $T$ and thus results in a very fast algorithm. Karaman and Frazzoli showed in [6] that the original RRT algorithm finds a suboptimal solution with probability one, and introduced the RRT* algorithm that uses local rewiring of the tree connections in order to converge to an optimal solution with probability one as the number of iterations goes to infinity. The RRT* algorithm [1] goes one step further by doing a cascaded rewiring in order to maintain optimality of the wiring of the tree at each iteration. Other variants, such as BIT [4] and RRTX [12], use batch processing or pseudo-optimality conditions in order to accelerate the execution of the algorithm but rely on the same algorithmic idea as in RRT*.

The previous algorithms have been proven to find an optimal solution as the number of samples tends to infinity, but convergence might be slow, especially in large search spaces. Several techniques have been proposed to alleviate this problem. Post-processing [10] of the solution is a common technique, but it only allows for local optimization of the current solution.

In order to converge to the optimal solution, the planning algorithm needs to always return a path near the optimal solution and in the same homotopy class, which cannot be guaranteed in general.

Given an admissible heuristic $h(x_1, x_2)$, that is, a lower-bound on the cost from $x_1 \in S$ to $x_2 \in S$, and assuming a solution from $x_{\text{start}}$ to $x_{\text{goal}}$ has cost $J$, the relevant region is defined by

$$\mathcal{X}_{\text{rel}}(J) = \{x \mid h(x_{\text{start}}, x) + h(x, x_{\text{goal}}) \leq J\}. \tag{2}$$

It is well known that the optimal solution will lie in $\mathcal{X}_{\text{rel}}(J)$, thus samples outside this region will not be near the optimal solution and will not help convergence. Two methods have been proposed to utilize this information: a) Rejection sampling, which samples points in $S$ but only keeps the points in $\mathcal{X}_{\text{rel}}(J)$; b) Informed sampling [5], which allows to directly sample in $\mathcal{X}_{\text{rel}}(J)$, and thus generates only good new samples, but it only works with specific forms of the heuristic, such as Euclidean distance. In addition, for these methods to work well, a good heuristic is required, and finding the best heuristic is often equivalent to solving the original problem.

In [7] the idea of displacing the new sample is used for systems with drift. A correction term is computed according to the vector field describing the drift of the system and then added to the new sample to reposition it in a lower cost region. Coming from a very different perspective, [2] shows how a path-planning problem can be solved as an optimization problem following a stochastic gradient descent. The algorithm applies to shortest Euclidean path and relies on the fact that for this problem, the optimal solution consists of an alternate of straight line segments in free space and geodesics on the obstacles boundaries. The solution can then be defined by the points of intersection between the geodesics and the straight lines, giving a parametrization of the solution. The gradient descent allows the minimization of the cost, while the use of intermittent diffusion allows to escape local minima in order to converge to a global solution.

In the rest of the paper, we first present a simple numerical experiment exhibiting the convergence rate of sampling-based
algorithms and perform a theoretical analysis of the probability of sampling relevant points as a function of the dimensionality of the search space. We then study how to optimize the location of the samples in order to minimize the error in the solution. We subsequently introduce the DRRT algorithm, which merges the optimization of sample position within the RRT framework. Finally, we compare the results of the DRRT algorithm against other similar state-of-the-art algorithms.

II. THE HYPERCUBE DIAGONAL EXPERIMENT (HDE)

In this section, we present a simple numerical experiment that will be used to compare results of algorithms converging to an optimal solution. Let the search space \( S \) be a hypercube of dimension \( d \) with each dimension taking values from \(-1\) to \(1\), namely \( S = [-1, 1]^d \). Assume momentarily that \( S \) is obstacle free. Let the starting point be

\[
x_{\text{start}} = [-1, -1, \ldots, -1],
\]

and the goal point be

\[
x_{\text{goal}} = [1, 1, \ldots, 1],
\]

that is, \( x_{\text{start}} \) and \( x_{\text{goal}} \) are the two opposite corners of the hypercube \( S \). The cost function \( c \) is the length of the path, normalized by \( 2\sqrt{d} \).

\[
c(x_1, x_2) = \frac{\|x_1 - x_2\|}{2\sqrt{d}}, \quad \forall x_1, x_2 \in S.
\]

Clearly, the optimal solution of the HDE is the straight line connecting \( x_{\text{start}} \) and \( x_{\text{goal}} \), that is, the diagonal of the hypercube. The length of the diagonal is \( 2\sqrt{d} \), so the cost of the optimal solution is \( c^* = 1 \). Using a normalized cost, such that the optimal solution is independent of the dimension, allows us to easily compare the convergence results across multiple dimensions.

In order to study convergence, we enforce a maximal distance between consecutive elements of the solution path \( \pi \). In the RRT family, this parameter is often called the range of the algorithm and it is used as the maximum length allowed when creating a new edge in the tree \( T \). Let the range for the HDE be

\[
\text{range} = 0.1\sqrt{d}.
\]

Similarly to the cost, the range is normalized, such that the optimal solution can be built with the same number of nodes, independently of the dimension \( d \). Specifically, an optimal solution can be built using exactly 19 nodes spread uniformly between \( x_{\text{start}} \) and \( x_{\text{goal}} \) for any dimension \( d \), as can be seen in Figure 1.

III. PURE SAMPLING STRATEGY

In this section, we analyze the probability of sampling “good” points using a uniform sampling strategy. A “good” sample is a point of the search space that is likely to help the convergence of the algorithm, that is, the point has to be sampled close to the optimal solution. Note that for real applications, the optimal solution is unknown, so it cannot be used to generate samples.

Let \( \pi^* \) be the optimal path from \( x_{\text{start}} \) to \( x_{\text{goal}} \) and let \( \varepsilon > 0 \) be a distance threshold. Define a tube for the path \( \pi^* \) and radius \( \varepsilon \) to be the set of points within a distance \( \varepsilon \) from \( \pi^* \),

\[
tube(\pi^*, \varepsilon) = \left\{ x \in S \mid \exists i \in [1, N], \text{distance}(x, [\pi^*_{i-1}, \pi^*_i]) \leq \varepsilon \right\},
\]

where \( \text{distance}(x, [a, b]) \) is the shortest distance between the point \( x \) and the segment \([a, b]\). Let also \( x_{\text{rand}} \) be a random variable, uniformly distributed over the sampling space \( X_s \). Define the event \( GS \) (“good” sample) to be the event that \( x_{\text{rand}} \) is within \( \varepsilon \) from the optimal path, that is,

\[
GS = \{ x_{\text{rand}} \in \tube(\pi^*, \varepsilon) \}.
\]

The probability of sampling a “good” point is then equal to the ratio of the volume of \( \tube(\pi^*, \varepsilon) \) and the volume of \( X_s \),

\[
P(GS) = \frac{|\tube(\pi^*, \varepsilon)|}{|X_s|}.
\]

Consider now the case of the HDE with a uniform sampler within the hypercube. Since the optimal solution \( \pi^* \) is the diagonal of the hypercube and the sampling space \( X_s \) is the entire hypercube, it follows that

\[
P(GS) = \frac{2\sqrt{d}V_{d-1}(\varepsilon)}{2^d} = \frac{2\sqrt{d}\varepsilon^{d-1}V_{d-1}(1)}{2^d} = \left( \frac{\varepsilon}{2} \right)^{d-1} \sqrt{d}V_{d-1}(1),
\]

where \( V_d(r) \) is the volume of a \( d \)-ball of radius \( r \). It can be seen that \( \sqrt{d}V_{d-1}(1) \) reaches the maximum of \( \sqrt{d} \) for \( d = 4 \), so

\[
P(GS) \leq \frac{\sqrt{d}\pi^{3}/6}{\left( \frac{\varepsilon}{2} \right)^{d-1} \sqrt{d}V_{d-1}(1)}.
\]

The dominant term in (13) is \( \varepsilon^{d-1} \). Hence, as \( \varepsilon \) gets smaller, the probability of sampling “good” points decreases, especially when \( d \) is large.

![Fig. 1: The Hypercube Diagonal Experiment in three dimensions. The red line shows the optimal solution and the stars (+) show the nodes for the optimal solution with the minimal number of nodes.](image)
If a heuristic \( h \) is available, smarter sampling can be performed. In particular, sampling can be done only in the relevant region, \( \mathcal{X}_r = \mathcal{X}_{\text{rel}}(J) \). Assuming \( J^* \) is the cost of the optimal solution, we have \( J^* \leq J \) and thus \( \mathcal{X}_{\text{rel}}(J^*) \subseteq \mathcal{X}_{\text{rel}}(J) \) and \( |\mathcal{X}_{\text{rel}}(J^*)| \leq |\mathcal{X}_{\text{rel}}(J)| \).

Assume \( |\mathcal{X}_{\text{rel}}(J^*)| > 0 \). If \( |\mathcal{X}_{\text{rel}}(J^*)| \) was equal to 0, the optimal value function would already be known along the optimal path and the problem would thus already be solved. Then

\[
P(GS) = \frac{2\sqrt{dV_0-1(\varepsilon)}}{|\mathcal{X}_{\text{rel}}(J)|} \leq \frac{2\sqrt{dV_0-1(\varepsilon)}}{|\mathcal{X}_{\text{rel}}(J^*)|} \leq \frac{\sqrt{7\pi^3}}{3|\mathcal{X}_{\text{rel}}(J^*)|^{d-1}}.
\]

Thus, if a heuristic is known, smart sampling can be used to improve the probability of “good” samples, but the dominant term is still \( \varepsilon^{d-1} \).

For instance, with \( \varepsilon = 0.2 \) in the case of two dimensions, the probability upperbound is around 20%, so “good” samples are likely, but in dimension 10, the probability upperbound is \( 10^{-9}\% \) so “good” samples become a very rare event.

Moreover, we see an exponential increase in the number of iterations required for convergence, which corresponds to the exponential decrease in the probability of sampling good points seen in \( (16) \).

IV. Optimizing Sample Location

In this section, we analyze how samples can be repositioned in order to minimize the cost without adding more samples, and thus increase the convergence rate. The idea is to minimize the error of the estimated value function with a fixed number of samples. In our case, the value function at \( x \in S \) is the optimal cost to go from \( x_{\text{start}} \) to \( x \).

Suppose a certain number of samples has been drawn and the information gathered has been stored in a tree \( \mathcal{T} \). The tree is a set of nodes with root node, \( x_{\text{start}} \), corresponding to \( x_{\text{start}} \) in the search space. Each node \( n \in \mathcal{T} \) contains the following information:

- \( x_n \): the corresponding state in the search space \( S \),
- \( p_n \): the parent node of \( n \) in the tree,
- \( C_n \): the list of node children of \( n \),
- \( c_n \): the cost from \( x_{\text{start}} \) to node \( n \) following the strategy encoded by \( \mathcal{T} \),
- \( N_n \): the list of nodes that are the nearest neighbors of \( n \) in the search space.

Let \( V(x) \), for \( x \in S \) be the value function we want to estimate. That is, \( V(x) \) is the minimum cost to reach \( x \) starting from \( x_{\text{start}} \). Let \( S^* \) be the subset of \( S \) reachable from \( x_{\text{start}} \), that is,

\[
V(x) = \begin{cases} 
\infty, & \forall x \in S \setminus S^* \\
\min_{n \in S^*} c_n + c(n, x), & \text{isFeasible}(n, x) \\
\infty, & \text{otherwise,}
\end{cases}
\]

where \( \text{isFeasible}(n, x) \) is true if the path from \( n \) to \( x \) is obstacle-free. In other words, \( \hat{V} \) is defined at the location of the samples of the tree by their cost value, and extended to the entire search space using a feasible path connecting to the tree with lowest cost. By construction, all trajectories in \( \mathcal{T} \) are feasible, thus

\[
\hat{V}(x) \geq V(x),
\]

because \( V \) is the minimum cost to reach \( x \).

The problem is to find the best location for the samples, organized in \( \mathcal{T} \), such that \( \hat{V} \) estimates \( V \) as closely as possible on \( S^* \). The problem can be formulated as

\[
\arg\min_{x_n, n \in \mathcal{T}} \frac{1}{|S^*|} \int_{S^*} |\hat{V}(x) - V(x)|dx \quad (20)
\]

\[
\Leftrightarrow \arg\min_{x_n, n \in \mathcal{T}} \frac{1}{|S^*|} \int_{S^*} |\hat{V}(x) - V(x)|dx \quad (21)
\]

\[
\Leftrightarrow \arg\min_{x_n, n \in \mathcal{T}} \frac{1}{|S^*|} \int_{S^*} \hat{V}(x)dx. \quad (22)
\]

Fig. 2: Time and number of iterations to converge within 2.5% of the optimal cost for the hypercube diagonal experiment as a function of the dimension of the problem.
Equation (19) is used to go from (20) to (21), and noticing
that the integral of $\hat{V}$ over $S^*$ is a constant independent of
the samples allows us to simplify the problem to (22).

Integrating $\hat{V}$ over $S^*$ is computationally very expensive,
but it can estimated using the samples of the tree, where $\hat{V}$
can be evaluated from,
\[
\frac{1}{|S^*|} \int_{S^*} \hat{V}(x) dx \simeq \frac{1}{|T|} \sum_{n \in T} \hat{V}(x_n) = \frac{1}{|T|} \sum_{n \in T} c_n. \tag{23}
\]
Since $|T|$ is constant, the optimization problem is reduced to
\[
\min_{x_n, n \in T} \sum_{n \in T} c_n. \tag{24}
\]
By the construction of $T$, we have
\[
c_n = \begin{cases} 
0, & \text{if } n = \text{start}, \\
(c(x_{pa}, x_n) + c_{pa}), & \text{if } n \neq \text{start}. 
\end{cases} \tag{25}
\]
We can then compute
\[
\sum_{n \in T} c_n = \sum_{i \in T} \sum_{j \in T} nb_{path}(i, j)c(x_i, x_j). \tag{26}
\]
where $nb_{path}(i, j)$ is the total number of paths in the tree
using the $(i, j)$ edge. Because of the tree structure, there is
exactly one path connecting the root of the tree to each node.
Moreover, this is true for any subtree, so
\[
nb_{path}(i, j) = \begin{cases} 
0, & \text{if } p_j \neq i \\
1 + nb_{des}(j) = d_j, & \text{otherwise}, 
\end{cases} \tag{27}
\]
where $nb_{des}(j)$ is the number of descendants of $j$ in $T$. The
paths going through the edge from $i$ to $j$ are the path to $j$ plus
the paths to all the descendants of $j$. We thus get,
\[
\sum_{n \in T} c_n = \sum_{i \in T} \sum_{j \in C_i} d_j c(x_i, x_j). \tag{28}
\]
By taking the gradient of the previous expression, we find
the direction that minimizes the error on the value function as follows
\[
\frac{\partial}{\partial x_i} \sum_{n \in T} c_n = \sum_{j \in C_i} d_j \frac{\partial c}{\partial x_i}(x_i, x_j) + d_i \frac{\partial c}{\partial x_i}(x_{pa}, x_i). \tag{29}
\]
This gradient with respect to the position of the node $i$ in
the search space is easily computable, as it depends only on
the parent node $p_i$ and the children nodes $C_i$.
A gradient descent can then be used in order to optimize
the position of the samples as long as the tree $T$ stays valid,
that is, every edge of $T$ stays feasible and obstacle-free.
This operation can be seen as an a-posteriori informed
sampling in the sense that the sample positions are updated
after having been added to the tree.

![Fig. 3: Computation of the gradient using local information -
Gradient corresponding to the children and the parent node in
green, and total gradient in red.](image)

V. OPTIMIZING SAMPLE LOCATION - RESULTS

Recall the minimization problem, $\min_{n, n \in T} \sum_{n \in T} c_n$ and
note that a trivial solution of this problem is to move every
single node to $x_{start}$, reducing the cost of each node to 0. This
solution is of no interest, so the following constraints need to
be added to the problem:

- The start and goal nodes cannot be repositionned as they are part of the definition of the problem.
- The coverage of the search space should remain unchanged because we do not want to lose the information that has been acquired about the search space through sampling. Leaf nodes of the tree will then be fixed in the search space.

Define the interior nodes of the tree as the set of nodes that are
not leafs of the tree and are different than start and goal. The
gradient descent is then applied only to the interior nodes of the tree.

Algorithm 1: Gradient Descent

1. while termination condition is not reached do
   2. foreach $n \in T$ do
      3. if $n == \text{start} \ OR \ n == \text{goal} \ OR \ isLeaf(n)$ then
         4. continue;
      5. $x_{temp} \leftarrow x_n - \alpha \frac{\sum_{n \in T} c_n}{\sum_{n \in T} \partial x_i} c_n$;
      6. if isFeasible($x_{pa}, x_{temp}$) AND
         7. isFeasible($x_{temp}, C_n$) then
            8. $x_n \leftarrow x_{temp}$;

Algorithm 1 shows the structure of the gradient descent. Nothing happens to start, goal and the leaf nodes of $T$. For the other nodes $n$, a temporary location $x_{temp}$ is computed from $x_n$, following the gradient with a scaling factor $\alpha$. If the new location maintains feasibility of the edges of $T$, that is, connection between $n$ and its parent $p_n$ and connections between $n$ and its children $C_n$, the location of $n$ is updated. The process is iterated until a termination condition is reached, for example a finite number of iterations has been reached or a convergence criterion is satisfied.
Figure 4 shows the evolution of the tree during gradient descent. The first row shows the tree structure and the second row shows the position of the interior nodes of the tree. The interior nodes start with an almost uniform distribution over the search space, and as gradient descent is applied, they start to concentrate around the optimal trajectories in each homotopy class. As the gradient descent continues, at iteration 100, the samples get sparser even on the optimal trajectories and start accumulating at specific points of the environment, namely the corners of obstacles, as expected for this environment.

Figure 5 shows in greater detail the distribution of the interior nodes after applying gradient descent. Circled in red are the accumulation points, and red lines show the tree built with these accumulation points. This tree corresponds to the optimal tree starting from the bottom left corner built on the visibility graph of this environment. That is, the optimal solution from the bottom left corner to anywhere in the search space can be found using this tree and a new node positioned at the desired goal.

Without any assumption on the type of obstacles, the gradient descent recovered the important points of the environment. In particular, for a shortest Euclidean path with polygonal obstacles, we recovered, as expected, that these points are located at the corners of obstacles, thus finding the optimal tree on the visibility graph given the starting position $x_{\text{start}}$.

Figure 6 shows similar results for an environment with circular obstacles. In this case, the samples accumulate on the boundary of the obstacles, which are the points of interest when searching for the shortest path in that type of environment.

Figure 7 shows the convergence of Optimized Tree Cost and Goal Cost.
Figure 7 shows the evolution of the cost optimized by the gradient descent and of the best cost to the goal as a function of the number of iterations. Each iteration is composed of two steps:

- Optimize the wiring of the tree, given the samples (in a similar fashion than the LPA* algorithm [8]).
- Optimize the location of the samples, given the tree structure (gradient descent).

The cost after each step is shown on the plots. We can see an overall reduction of the cost for the entire environment, as well as a reduction of the cost to the goal. The goal cost does not show a smooth decrease because it is not the value being optimized, and moving a node that is part of the best path can temporarily increase the cost.

VI. THE DRRT ALGORITHM

In this section we present the Deformable Rapidly-Exploring Random Trees algorithm that merges the idea of optimizing the position of the samples in the framework of RRT-like algorithms.

When samples are added iteratively, there is no need to do a global gradient descent as changes occur only locally. Suppose the tree is already optimized with respect to both wiring of the tree and position of the samples. Then adding a new sample only creates local changes near that sample, and perhaps also down the branch of the tree, if it can be used to improve the cost of other nodes. The branch from the root of the tree to the new sample can be optimized, and thus improve the value function at the new location, before using it to propagate information through the tree.

Each iteration of the DRRT algorithm can be described in three main steps:

- Sample a new point \( x_{\text{new}} \) and connect to the tree as a leaf.
- Optimize the location of the nodes on the branch from the root to \( x_{\text{new}} \).
- Propagate changes through the tree.

The core of the algorithm, shown in Algorithm 2, is similar to the one of RRT* with an extra step of performing gradient descent. At each iteration, the queue \( Q \), used to order the nodes to be updated, is cleared. A new sample \( x_{\text{new}} \) is drawn, and the nearest element \( x_{\text{near}} \) in \( T \) is found. The algorithm steers \( x_{\text{new}} \) to be within the defined maximum range of \( x_{\text{near}} \). If the path from \( x_{\text{near}} \) to \( x_{\text{new}} \) is feasible, a new node \( n_{\text{new}} \) is created and added to the tree \( T \). The node is initialized with \( x_{\text{near}} \) has its parent and its neighbors are computed. Within the neighbors, the one that minimizes the cost is chosen and the node is updated.

Algorithm 2: The DRRT Algorithm

1. \( T = \{ \text{start} \} \);
2. while termination condition is not reached do
3. \( Q \leftarrow \emptyset \);
4. \( x_{\text{rand}} \leftarrow \text{New Sample}() \);
5. \( x_{\text{near}} \leftarrow \text{Near}(x_{\text{rand}}) \);
6. \( x_{\text{new}} \leftarrow \text{Steer}(x_{\text{rand}}, x_{\text{near}}) \);
7. if isFeasible\((x_{\text{near}}, x_{\text{new}})\) then
8. \( n_{\text{new}} \leftarrow \text{New}\_\text{Node}(x_{\text{new}}, x_{\text{near}}) \);
9. \( \text{Optimize\_Parent}(n_{\text{new}}) \);
10. \( b \leftarrow \text{Branch}(n_{\text{new}}) \);
11. \( \text{Gradient\_Descent}(b) \);
12. \( Q.\text{insert}(b) \);
13. \( \text{Propagate\_Changes}() \);

Function 1: Optimize\_Parent\((n)\)

1. \( p_n \leftarrow \arg \min_{n_{\text{nbh}} \in N_n} c(x_{\text{nbh}}, x_n) + c_{\text{nbh}} \);
2. \( c_n \leftarrow \min_{n_{\text{nbh}} \in N_n} c(x_{\text{nbh}}, x_n) + c_{\text{nbh}} \);

Function 2: Gradient\_Descent\((b)\)

1. while termination condition is not reached do
2. \( \text{foreach } b_i \in b \text{ do} \)
3. \( t \leftarrow 1 \);
4. while \( J_V(x_{b_i} - t \frac{\partial J_V}{\partial x_{b_i}}) > J_V(x_{b_i}) - \frac{1}{2} \| \frac{\partial J_V}{\partial x_{b_i}} \|^2 \) do
5. \( t \leftarrow \beta t \);
6. temp \( \leftarrow x_{b_i} - t \frac{\partial J_V}{\partial x_{b_i}} \);
7. if isFeasible\((x_{p_i}, \text{temp})\) AND isFeasible\((\text{temp}, c_{b_i})\) then
8. \( x_{b_i} \leftarrow \text{temp} \);
9. \( \text{foreach } b_i \in b \text{ do} \)
10. if \( x_{b_i} \) has changed then
11. Update \( b_i \) in nearest neighbor data structure;

The core of the DRRT algorithm involves the creation of a branch \( b \) by following parent pointers up to the start node. This branch contains every node encountered except for \( n_{\text{new}} \) and \( \text{start} \). These nodes are excluded because we want to ensure that their position is not changed. Moreover excluding these two nodes guarantees that every element of the branch has both a parent and at least one child.
by cost + heuristic and the element with the smallest key is treated at each iteration. The element is tried as a parent candidate for all its neighbors and is chosen if it improves the cost. The cost is propagated to all its children, and the modified nodes are added to the queue. The process stops when all nodes in the relevant region \( X_{el} \) are processed, that is, when the smallest element of the queue has a key larger than the goal.

\[
\begin{align*}
\text{Function 3: Propagate\_Changes} & \\
\text{while } Q \text{ is not empty do} & \\
\quad cl & \leftarrow Q.\text{pop\_min}(); \\
\quad \text{if } \text{Key}(cl) > \text{Best\_Cost} & \\
\quad \quad \text{break; } & \\
\quad \text{foreach } nbh \in X_{el} & \\
\quad \quad \text{if } c_{el} + c(x_{el}, x_{nbh}) < c_{nbh} & \\
\quad \quad \quad c_{nbh} & \leftarrow c_{el} + c(x_{el}, x_{nbh}); \\
\quad \quad \quad p_{nbh} & \leftarrow cl; \\
\quad \quad \quad Q.\text{update}(nbh); & \\
\quad \text{foreach } c \in C_{el} & \\
\quad \quad c & \leftarrow c_{el} + c(x_{el}, x_{c}); \\
\quad \quad Q.\text{update}(c); & \\
\end{align*}
\]

VII. NUMERICAL RESULTS

A. The Hypercube Experiment

In this experiment, we use the HDE and run the algorithms for different dimensions of the problem until the cost is within 3% of the optimal solution. The DRRT algorithm was implemented in the Open Motion Planning Library (OMPL) [13], which provides efficient implementations of the state-of-the-art sampling-based path-planning algorithms, and can be easily integrated with many robotic systems and simulation environments. The DRRT algorithm is compared here against RRT* and RRT#. For all three algorithms we used the regular algorithm as well as rejection sampling and informed sampling variants.

In each dimension, and for each variant, 10 simulations were run and the mean and standard deviation were computed. These are shown in Figure 8. We see here again, a very minimal influence of the sampling variants. In lower dimensions, the probability of sampling good points is high enough that both RRT* and RRT# find a near optimal solution faster than DRRT. But the time to find a near optimal solution for those two algorithms increases extremely fast with the dimension of the problem, whereas for the DRRT algorithm, the time is almost constant. Thus, the DRRT algorithm is faster on this problem by multiple orders of magnitude for any dimension larger than three.

Figure 9 shows the results of the same experiments, only for the plain variant of each algorithm, in terms of iterations. In dimension two, all algorithms need the same number of nodes to solve the problem, but as the problem dimensionality grows, this number stays almost constant for the DRRT algorithm, whereas it increases exponentially for the RRT* and the RRT# algorithms.

B. 6DOF Manipulator

The algorithm was used and benchmarked using the OMPL library [11] against the current state-of-the-art in the V-REP [3] simulation environment on the Mico robotic arm from Kinova Robotics. Figure 10 shows the simulation environment. The robot on the right was used for the benchmarking and the trajectories found were used to manipulate the cups. The arm has six degrees of freedom, and the algorithm was benchmarked on three sets of start and goal positions that can be seen in Figure 10. For each set, each algorithm was run for 60 seconds and the evolution of the cost over time for each run was reported.

Figure 11 shows the mean of the best cost over all trajectories, as well as the standard deviation. We can see that RRT* finds the largest cost, followed by RRT# and RRTX closely together. All three algorithms have a similar standard deviation. Three variants of the DRRT algorithm were tested,
the standard algorithm DRRT, a delayed optimization version DRRTd (no optimization is performed until the first solution is found) and a variant where the node optimization is done only 30% of the time, DRRT0.3. The three variants gave significantly better results than the other three algorithms, both in terms of the mean and the standard deviation of solution.

Figure 11 shows the cost as a function of the convergence time, the time after the first solution to the goal was found. This figure allows us to analyze how the algorithms converge once a feasible solution has been found. We can see for RRT*, RRT# and RRTX that the convergence rate is small, due to the unlikeliness of sampling "good" points in high dimensions. The three DRRT variants show much lower cost for the first solution, and the convergence rate is significantly larger.

The solutions found with DRRT are better, but this improvement comes at a cost. More exploitation of the data means less exploration and thus the DRRT algorithm is slower to find a first feasible solution. Figure 13 shows the percentage of solutions found by those algorithms over time. A total of 100 problems were solved for each pair of start and goal position. We can see that RRT*, RRT#, RRTX and DRRTd quickly find a feasible solution in all cases, but the DRRT and the DRRT0.3 variants are much slower and do not always find a feasible solution within the allocated 60 seconds.

As expected, the ordering of the algorithms by best cost is in the reverse order of the algorithms by time to find the first feasible solution. There is a compromise between time it takes to find the first solution and optimality of the solution found. These results suggest that, if time is not a constraint, the DRRT algorithm is the best option as it has the fastest convergence and the lower cost once a solution is found. In the case of limited computational time, a hybrid approach like the DRRTd variant allows to quickly find a feasible solution before spending computational time in optimization.

VIII. CONCLUSION

In this paper, we introduced a new experiment exhibiting the convergence characteristics of sampling-based planning algorithms as a function of the dimension of the search space of the problem. This experiment was then used to show the limits of uniform sampling as the dimension grows. We introduced a way to optimize the sample location in the search space in order to optimize the cost with a given number of samples. Finally, this optimization was integrated in the RRT framework to create the DRRT algorithm, whose results significantly outperform the state-of-the-art sampling algorithms.
REFERENCES


