Motion Planning for Highly Constrained Spaces

Anna Yershova and Steven M. LaValle

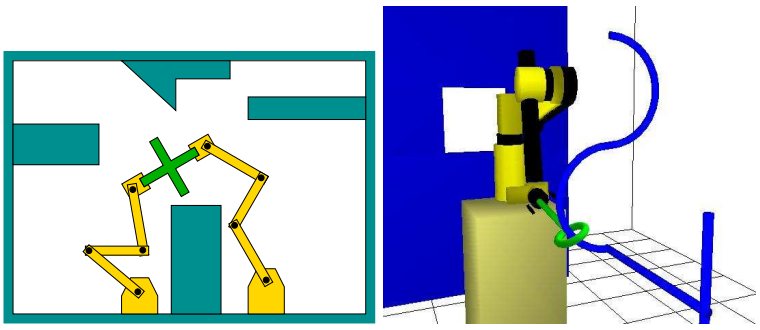
Abstract We introduce a sampling-based motion planning method that automatically adapts to the difficulties caused by thin regions in the free space (not necessarily narrow corridors). These problems arise frequently in settings such as closed-chain manipulators, humanoid motion planning, and generally any time bodies are in contact or maintain close proximity with each other. Our method combines the aggressive exploration properties of RRTs, the adaptive sampling domain of DDR-RTs, and the intrinsic dimensionality-reduction properties of kd-trees to focus the sampling and searching in the appropriate subspaces. We handle closed-chains and other kinds of constraints in a general way that avoids inverse kinematics computations, if desired. We have implemented the method and show its computational advantages on a variety of challenging examples.

1 Introduction

In many motion planing problems, the feasible subspace becomes thin in some directions. This is often due to kinematic closure constraints, which restrict the feasible configurations to a lower-dimensional manifold or variety. This may also be simply due to the way the obstacles are arranged. For example, is planning for a closed chain different from planning a sliding motion for a washer against a rod? An illustration for the two instances of such problems is shown on Figure 1. Traditionally, the two problems are solved with different methods in motion planning. However, the two seemingly different problems have similar algebraic and geomet-

Anna Yershova
Department of Computer Science, Duke University, Durham, NC 27707, USA, e-mail: yershova@cs.duke.edu

Steven M. LaValle
Department of Computer Science, University of Illinois, Urbana, IL 61801 USA, e-mail: lavalle@cs,uiuc.edu



(a) (b)

Fig. 1 (a) Manipulation problem for a closed chain. (b) Manipulation problem for sliding a washer against a rod.

ric structure of the sets of feasible configurations. In either case, there may exist functions of the form

$$|f_i(q)| \leq \varepsilon_i, \varepsilon_i \geq 0$$

that contain most or all of the feasible set. Typically, ε_i is very small, and in the case of closed chains, $\varepsilon_i = 0$. When this occurs, a region of the feasible space has small *intrinsic* dimensionality, in comparison to the *ambient* configuration space. Note that the f_i are not necessarily given.

In automated manufacturing, and manipulation planning problems similar to the two discussed above are abundant. In some cases the functions f_i are explicitly provided. A large set of mechanisms, such as PUMA robots, or humanoid robots are well studied, with readily available expressions for the forward kinematics, and grasping configurations. Many practical solutions exists for solving such problems. Very often, however, the expression for the constraints is not readily available, such as for the example of the washer and the rod. There are fewer methods available for such problems, unless the constraints are explicitly modeled.

There are thousands of CAD models of mechanisms, and objects (including rods) of different shapes, designed for manipulation systems in industrial robotics applications; it is frustrating that no technique exists for solving these problems in a unified way. Users are faced with modeling issues after differentiating the problems into classes.

Planning for closed chains is usually considered as a separate class of problems, since the kinematic constraints are given a priory. Analytical approaches construct explicit geometrical and topological representation of the closure set [4, 13, 9], but are usually inefficient in practice. Practical sampling-based methods [16, 3] usually project the closure set on the subset of parameters, on which the planning is performed [8, 5]. An inverse kinematics solver is used in these approaches as a black box to get the solution back on the configuration space. These approaches are quite

successful in practice, however, there are disadvantages associated with using inverse kinematics:

- Analytical solutions for inverse kinematics for arbitrary manipulators are prohibitively complex and can only be applied to relatively simple mechanisms with low number of degrees of freedom [6].
- Numerical techniques exist for solving inverse kinematics for arbitrary mechanisms [15]; however, they are not presently at the stage of being applied in practice.
- Even if inverse kinematics solutions were readily available, the choice of the subset of parameters on which projection is made may significantly affect the performance of a planner. Planning becomes inefficient around singularities associated with the projection on the chosen subset of parameters. There is no general method, though, for the choice of parameters. In practice, a careful analysis of the linkages is usually required before the method is applied [5].

Separate techniques are also developed for motion planning problems with constrained geometries, such as the example of sliding the washer against the rod. Several approaches were proposed recently [7, 17] for solving such problems. The DDRRT method is especially promising, since it allows handling both constrained geometries and closed kinematic chains in a unified way. However, in its original presentation it can only handle a set of motion planning problems in low dimensional configuration spaces.

In this paper we propose an extension of the DDRRT motion planner which can handle planning under the set of constraints of the form $|f_i(q)| \leq \varepsilon_i$. It needs neither an inverse kinematics solver, nor an explicit expression for f_i, which provides maximum flexibility to the user. The method keeps the representation of the set of feasible configurations (dynamic sampling domain) in the kd-tree data structure. Kd-trees are well known for the ability to capture intrinsic dimensionality of the subsets in an ambient space. The planning is done locally using the RRT planner [12] inside the represented portion of the feasible set. Our method combines the aggressive exploration properties of RRTs with the intrinsic dimensionality-reduction properties of kd-trees to focus the sampling and searching only on the feasible set inside the configuration space.

We have tested the algorithm on a dozen problems, including both basic motion planning and planning under closed-chain kinematic constraints. We observed that the performance of the algorithm improves over the original RRT planner on most of the problems, often by an order of magnitude. In rare cases, when there is no improvement in the running time, the proposed method works only slightly worse than the RRT. We believe that designing a single algorithm that reliably performs on a large set of problems is important to the motion planning community.

It is important to note that the implementation of the algorithm presented in the paper needs the dynamic update performed on kd-trees. Even though there are well known algorithms for doing this citeOveVan82, according to our best knowledge, there is no available software with the dynamic kd-tree implementation, and more-

over, such implementaion is not straightforward. Therefore, our current work has additional value of being the first efficient implementation of the dynamic kd-trees.

In Section 2 we define the planning problem we consider. Sections 3 and 4 introduce the planner we propose, and the kd-tree data structure it uses, respectively. We show experimental results in Section 5, followed by conclusions and future work in Section 6.

2 Problem Description

We start with the description of the class of problems we consider in this paper.

2.1 The Motion Planning Problem

Let $W \subset \mathbb{R}^N$, $N = 2$ or 3, be the workspace, to which the robot and obstacles belong. Consider the *configuration space* C. The set of all of the configurations satisfying the constraints $C_{con} = \{q : |f_i(q)| \leq \varepsilon_i\}, \varepsilon_i \geq 0, i \geq 0$, is called the *constrained space*. The *free space*, C_{free}, is defined as the configurations $q \in C$, which satisfy the collision constraints. The *valid space* is the closure of the free space, $C_{val} = C_{free} \cup \partial C_{free}$. The *feasible space* is defined as $C_{fea} = C_{con} \cap C_{val}$; it contains the configurations that satisfy the constraints and avoid penetration into obstacles.

The motion planning problem is defined on C_{fea}. Given initial and goal configurations $q_{init}, q_{goal} \in C_{fea}$, find a continuous path $\tau : [0,1] \rightarrow C_{fea}$, such that $\tau(0) = q_{init}$, and $\tau(1) = q_{goal}$.

It is important to note that when $i = 0$ and $C_{fea} = C_{val}$ the feasible space may have similar topological properties to the feasible spaces for which the functions f_i are defined. In cases when $i = 0$, the obstacles of the configuration space may define the constraints similar to $|f_i(q)| \leq \varepsilon_i$ implicitly.

2.2 Special Case: Linkages with Closed Chains

One instance of the set of problems defined above can be obtained when planning for linkages with closed kinematic chains. Consider a *chain* of n links, such that each *link* L_i is a rigid body. When two links, L_i and L_j, are attached to each other, the place at which they attach is called the *joint* $J(L_i, L_j)$. Call L the collection of all of the links in the chain, and J the collection of all of the joints. The *underlying graph*, $G(J, L)$, in which the vertices correspond to all of the joints, and the edges are the corresponding links, represents the topology of the linkage. The underlying graph has cycles if and only if the linkage contains closed chains.

Each joint $J(L_i, L_j)$ carries information about the type of the attachment (revolute, spherical, etc.) This is often expressed as the homogeneous transformation matrix from the coordinate frame of one link to the frame of another. The variables in the matrix express the freedom of movement of the link around the joint with another link. This leads to a parametrization of the linkage (for example, the Denavit-Hartenburg representation [10]).

Setting each of the parameters to a fixed value results in a real-valued vector q, which represents a fixed configuration of the linkage. If $G(J,L)$ contains cycles, then not all of the configuration q yield an acceptable position and orientation of each of the links in the chain. Only configurations q which satisfy the closure constraints of the form $f_i(q) = 0$ result in valid configurations of the linkage. The closure constraints can be obtained by writing down two homogeneous transformation matrices for a coordinate frame of a link in each loop of a closed chain. Each of the matrices corresponds to the two different paths to the link alone the loop. The closure constrain can then be obtained by forcing the frame of the link to be the same, regardless of the path that was chosen.

Since the configurations satisfying the closure constraints $f_i(q) = 0$ are defined implicitly, they often can not be obtained analytically. It is natural to assume, therefore, that some numerical error, ε, is allowed for the configurations on the closure set. The value for ε is usually chosen based on the particular application. In the rest of the paper we consider, therefore, the relaxation of the closure constraints for closed chains $f_i(q) = 0$ to $|f_i(q)| \leq \varepsilon$.

3 Dynamic Domain RRT Planner

The RRT planner [12] was successfully used for many motion planning problems. It has a natural ability to aggressively explore the free configuration space. RRTs can be straightforwardly applied to the feasible spaces, by ensuring that the constraints $|f_i(q)| \leq \varepsilon_i$ are validated for each vertex in the tree. This approach is a baseline algorithm for our experiments. We describe it in details in Section 3.1.

It was shown in [17] that the RRT suffers from the local minimum problem in case when the free space is significantly smaller that the configuration space. The approach to overcome this difficulty in [17] was to maintain a local representation, called *dynamic domain*, of the explored portion of the free space. This allows the RRT planner to concentrate the search on the "useful" portion of the configuration space. The data structure used to maintain the dynamic domain in [17] could only handle up to six dimensional configuration spaces. We propose using kd-tree based representation to maintain dynamic domains in feasible configuration spaces. Kd-trees [1] can handle up to 25-50 degrees of freedom and millions of nodes, are able to significantly speed up nearest-neighbor calculations in motion planning applications [2], and have intrinsic dimensionality reduction abilities. We describe the dynamic-domain RRT in Section 3.2, and the kd-tree data structure for the dynamic domain in Section 4.

BUILD_RRT(q_{init})
1 $\mathscr{T}.init(q_{init})$
2 **for** $k = 1$ **to** K **do**
3 $q_{rand} \leftarrow$ RANDOM_CONFIG(C)
4 $q_{near} \leftarrow$ FIND_NEAREST_NEIGHBOR($q_{rand}, \mathscr{T}$)
5 **if** CONNECT($\mathscr{T}, q_{rand}, q_{near}, q_{new}$)
6 $\mathscr{T}$.add_vertex(q_{new})
7 $\mathscr{T}$.add_edge(q_{near}, q_{new})
8 Return $\mathscr{T}$

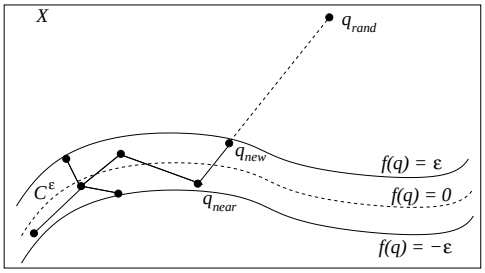


Fig. 2 The RRT-CONNECT construction algorithm.

BUILD_DDRRT(q_{init})
1 $\mathscr{T}.init(q_{init})$
2 **for** $k = 1$ **to** K **do**
3 $q_{rand} \leftarrow$ RANDOM_CONFIG(D)
4 $q_{near} \leftarrow$ FIND_NEAREST_NEIGHBOR($q_{rand}, \mathscr{T}$)
5 **if** CONNECT($\mathscr{T}, q_{rand}, q_{near}, q_{new}$)
6 $\mathscr{T}$.add_vertex(q_{new})
7 $\mathscr{T}$.add_edge(q_{near}, q_{new})
8 UPDATE(D)
9 Return $\mathscr{T}$

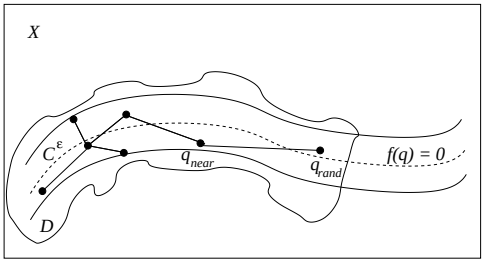


Fig. 3 The DDRRT-CONNECT construction algorithm.

3.1 The RRT Algorithm

Consider the pseudocode for building an RRT in C_{fea} shown on Figure 2. At iteration $k = 1$ an RRT contains only the initial configuration, q_{init}. At each iteration the

RRT grows, until either it contains the goal configuration (that is, a path from q_{init} to q_{goal} is found which is a branch in the RRT), or a limit on the number of iterations is reached. To grow the RRT, a random configuration, $q_{rand} \in C$, is chosen in Line 3. This configuration is not added to the tree, thus the constraints $|f_i(q)| \leq \varepsilon_i$ need not be satisfied at this step. In Line 4, the closest to q_{rand} configuration q_{near} from the nodes in RRT is selected. The connection from q_{near} to q_{rand} is attempted in Line 5. This corresponds to the interpolation between q_{near} and q_{rand}, such that the furthest configuration $q_{new} \in C_{fea}$ along the interpolation path from q_{near} is returned. If the interpolation step is successful, the new vertex q_{new}, and corresponding edge are added to the RRT. The function CONNECT performs interpolation and validation of the constraints $|f_i(q)| \leq \varepsilon_i$. The geometrical illustration of the algorithm is shown on Figure 2.

This algorithm requires neither parametrization of the points in C_{fea}, nor an inverse kinematics solver in case of planning for closed chains. This makes each line in the procedure very efficient. Given that n points were built by an RRT at a particular iteration, and the dimension of the configuration space C is d, the sampling step in Line 3 takes $O(d)$ time, and the nearest neighbor call in Line 4 takes $O(2^d n \log n)$ running time [2]. The computational time in this algorithm is not spent at any particular line of the pseudocode, but on the number of iterations needed for solving a problem. The drawback of this approach is similar to the one outlined in [17]. That is, the Voronoi bias of the points in the RRT determine the exploration behavior of the RRT. Since the sampling from the configuration space C in Line 3 does not take into account neither obstacles of configuration space, nor the constraints $|f_i(q)| \leq \varepsilon_i$, the same extensions are repeatedly attempted towards invalid configurations. This increases the number of iterations needed to solve a problem. The next section presents a better suited approach for constrained feasible spaces.

3.2 Dynamic Domain Sampling for RRT

To improve the Voronoi bias in the RRT exploration the approach in [17] proposes to maintain a dynamic domain D, that approximates the feasible configuration space with a simple shape, such as a collection of balls. This significantly reduces the effect of local minimum problem for RRTs. The sketch of the algorithm and its geometrical illustration are shown on Figure 3. The difference from the baseline RRT algorithm is in Line 3, in which dynamic domain is used for sampling instead of the configuration space. Assume again that n points were built by an RRT at a particular iteration, and the dimension of the configuration space C is d. For the kd-tree dynamic domain, the uniform sampling in Line 3 of the algorithm on Figure 3 is performed in $O(\log n)$ running time, the nearest-neighbor call in Line 4 requires $O(2^d n \log n)$ time, and the update function in Line 8 takes $O(\log n)$ time. Using the kd-tree data structure, the cost of each iteration is slightly increased, comparing to the original RRT algorithm on Figure 2. However, the number of iterations is usually reduced in the planning process, and therefore, the running time improves overall.

BUILD_KD_TREE(P, d, m, b, r)
Input: A set of points P, the dimension of the space d, the number of points to store in a leaf m, the bounding box b for P, and the thickness parameter r.
Output: The root of a kd-tree storing P

```
1  if P contains less than m points
2      return a leaf storing these points,
                 and an r-bounding box for P
3  else Split b into two boxes, b_1, b_2.
4      Find P_1 and P_2, the sets of the data points
       inside boxes b_1 and b_2.
5      v_1 =BUILD_KD_TREE(P_1, d, m, b_1, r)
6      v_2 =BUILD_KD_TREE(P_2, d, m, b_2, r)
7      Create node v storing the splitting plane l,
       splitting dimension k, bounding box b,
       and nodes v_1 and v_2, the children of v.
8      area = v_1.area + v_2.area
9      height = max(v_1.height,v_2.height) + 1
10     return v
```

Fig. 4 The algorithm for constructing a kd-tree for a set of points P.

The next section describes the implementation details for the kd-tree data structure to perform efficient sampling and update on the explored portion of the feasible configuration space.

4 Representing Feasible Configuration Spaces with KD-Trees

Consider the set S of data points lying inside a d-dimensional enclosing rectangle. We build the kd-tree inside this rectangle, and define it recursively as follows. The set of data points is split into two parts by splitting the rectangle that contains them into two children rectangles by a hyperplane, according to a specified rule; one subset contains the points in one child box, and another subset contains the rest of the points. The information about the splitting hyperplane and the boundary values of the initial box are stored in the root node, and the two subsets are stored recursively in the two subtrees. When the number of data points contained in some box falls below a given threshold, m, the node associated with this box is called a leaf node, and a list of coordinates for these data points is stored in this node.

We divide the current cell through the median of the points orthogonally to the cell's longest side. If there are ties then we select the dimension with the largest point spread. This ensures that the resulting kd-tree for n data points is balanced, with the height of the tree equal to $O(\log n/m)$.

We introduce the notion of *r-bounding rectangle* for a set of points P at a node m of the kd-tree, as an intersection of the bounding rectangle of the node m and a rectangle with the two opposite points p_1 and p_2 defined as $p_1 = (\min\{p_j^i | p^i \in P\}_{j=1}^d - \mathbf{r})$, and $p_2 = (\max\{p_j^i | p^i \in P\}_{j=1}^d + \mathbf{r})$. Here, r is the parameter that deter-

```
UPDATE_KD_TREE(p, d, m, v, b)
Input: The point to be added p, the number of points to store in a leaf m, the node v (initially the
root), and the bounding box b for v.
Output: The root of a kd-tree which includes P ∪ p
1  if v is a leaf
2       if (v.size > m)
3            BUILD_KD_TREE(v.P ∪ p, d, m, b, r)
4       else store p in v
5  else consider the two subboxes, b1, b2 of b
6       d1 = distance(p, b1)
7       d2 = distance(p, b2)
8       if d1 < d2
9            then v1 = UPDATE_KD_TREE(p, m, b1, v1)
10           else v2 = UPDATE_KD_TREE(p, m, b2, v2)
11      if (v1.height > 2v2.height) or
            (v2.height > 2v1.height)
12           BUILD_KD_TREE(v.P ∪ p, d, m, b, r)
13      area = v1.area + v2.area
14      height = max(v1.height,v2.height) + 1
15      return v
```

Fig. 5 The algorithm for updating the kd-tree with a new point p.

mines the *thickness* of the dynamic domain for the points in the RRT. This parameter is manually selected in our current implementation. However, for more effective performance, an automatic parameter tuning similar to [11] can be implemented.

The *kd-tree dynamic domain* is defined as the collection of all of the r-bounding rectangles of the points at the leaves of the kd-tree.

Next, we outline the four main functions for the kd-tree needed to implement the algorithm on Figure 3.

4.1 Construction

Our kd-tree is constructed using a recursive procedure, which returns the root of the kd-tree (see Figure 4). This construction algorithm is essentially identical to the case of constructing a kd-tree in an Euclidean space [14]. The differences in the implementation come from the need to maintain additional information, such as r-bounding rectangles, and the heights of the tree and each of the subtrees. This is needed for efficient sampling (Section 4.4) and update (Section 4.2) of the kd-tree.

The running time for building a kd-tree for a set of n points is $O(n \log n)$ [1].

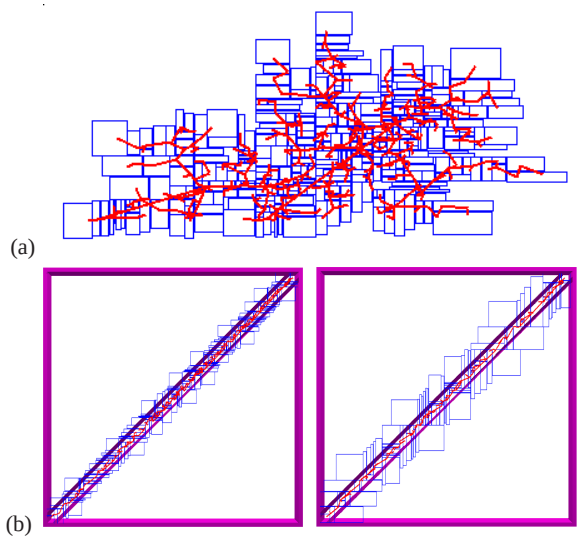


Fig. 6 (a) An RRT with 800 nodes in two dimensional space, together with the corresponding kd-tree dynamic domain is shown. (b) An RRT with 600 nodes is growing inside the diagonal corridor. The corresponding kd-tree dynamic domains with two different r-parameters are shown.

4.2 Dynamic Update

When a new point is added at line 8 on Figure 3, the kd-tree is updated according to the algorithm shown on Figure 5. First, the algorithm descends to a node, to which the new point belongs, and such that the height balance at line 10 of the algorithm would become invalid for the two children nodes after adding the point. Next, the construction procedure is called on this node. The area of the r-bounding boxes and the heights of the subtrees are then updated.

Given that there are n points in the kd-tree, the time to perform one update consists of $O(\log n)$ to descent to the node that needs rebalancing, and of $O(m \log m)$ to rebalance this node, in which there are m points associated with the node. The running time of the algorithm depends on the node which is rebalanced. Over many runs, the nodes with large number of points are rebalanced fewer times than the leaves of the tree. In fact, the node with n points in the worst case is rebalanced only during every n-th run. Therefore, the amortized analysis yields $O(\log n)$ worst case running time for this procedure.

Figure 6 (a) illustrates how the update procedure works, and how the corresponding binary tree looks like for 800 data points incrementally added to the tree. Figure 6 (b) shows two kd-trees for the set of points among the obstacles in configuration space. Each kd-tree in this figure corresponds to a different r-parameter.

```
KDTree::SAMPLE()
Output: sample q
1 q = root.SAMPLE()
2 return q
```

```
Node::SAMPLE()
1 p1 = v1.area/ area
2 p2 = v2.area/ area
3 with probability p1
4       q = v1.SAMPLE()
5 with probability p2
6       q = v2.SAMPLE()
```

```
Leaf::SAMPLE()
1 q =RANDOM_CONFIG(bnd_box)
```

Fig. 7 The algorithm portions for searching a kd-tree on the root level and internal and leaf nodes levels.

4.3 Nearest-Neighbor Query

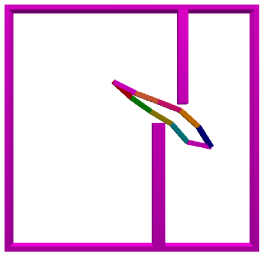
The query phase is performed identically to the procedure outlined in [1] and [2]. Therefore, we omit the discussion about it here. We only note that the query is performed in $O(2^d \log n)$ time [1], where d is the dimension of the configuration space C. This is the most expensive operation in a single iteration of the RRT algorithm on Figure 3.

4.4 Uniform Sampling

To sample the area represented by the kd-tree, the algorithm first descends to a leaf, with probability corresponding to the area of the r-bounding rectangle of the leaf. Next, it returns a sample from the r-bounding box of the leaf. The procedure is outlined on Figure 7. The running time of the procedure is $O(\log n)$, since only one descend alone the tree is needed. The resulting samples are guaranteed to be uniformly distributed over the collection of r-bounding boxes, since the boxes from different leaves do not intersect.

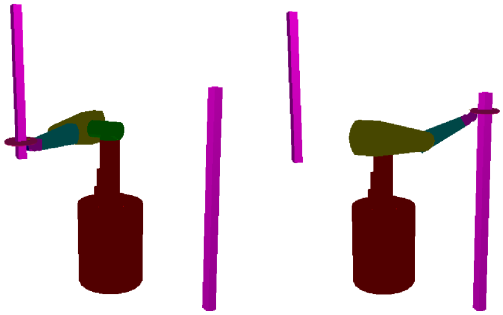
5 Experimental Results

We have compared the performances of the two RRT algorithms described in Figures 2 and 3. For each of the experiments, we show the running times, the number of nodes in the solution trees and the number of collision detection calls (CD) dur-



2dLoop3	RRT	DDRRT
time(sec)	170.70	5.5
nodes	4,347	881
CD calls	20,667	11,408

Fig. 8 The goal in this example is to move a closed chain with 12 links and revolute joints from the left to the right part of the environment through a narrow opening.



obelix	RRT	DDRRT
time(sec)	295.92	26.79
nodes	223	40
CD calls	817,900	100,642

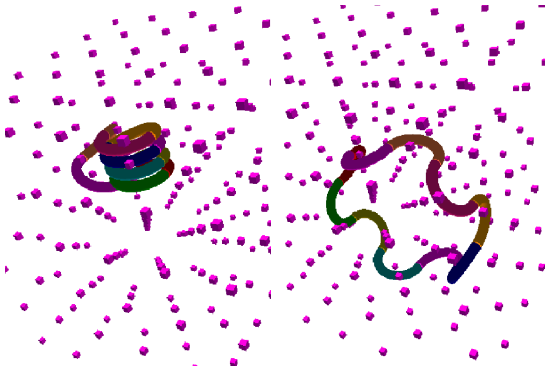
Fig. 9 The goal is to use the PUMA robot to move the toroidal object from one rod in the work space to the other.

ing the construction process averaged over 50 runs. The r-parameter was manually selected in these experiments.

The first experiment is for a 2d closed chain consisting of 12 identical rectangular links. All of the joints are revolute. The performance comparison of both of the RRT algorithms is shown on Figure 8. The improvement of the kd-tree-based approach is around 30 times over the original RRT in this example.

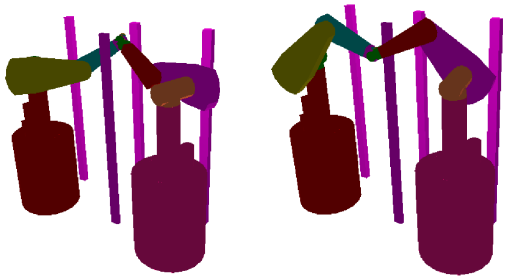
The second experiment does not involve closed chains explicitly (Figure 9). The goal in this example is to enable the PUMA robot to hold a toroidal object, and to move it from one rod in the work space to the other. Using the kd-tree approach there is an improvement of an order of magnitude in the running time for this experiment.

The third example involves unfolding a 3d closed chain with revolute joints and 10 degrees of freedom through a cloud of obstacles (Figure 10). The performance improvement in this experiment is not as significant as in the other examples. We speculate that adaptive tuning of the r-parameter is significant for solving the problem efficiently. There are several thin sheets in the configuration space of this prob-



3dLoop1	RRT	DDRRT
time(sec)	178.6	124.85
nodes	138	208
CD calls	4,143	2,481

Fig. 10 The goal in this example is to unfold a 3d-closed chain with 12 links amidst the cloud of obstacles.



3dtree2	RRT	DDRRT
time(sec)	240.58	35.81
no.nodes	869	843
CD calls	2,309	1,919

Fig. 11 This experiment involves two PUMA robots holding an object, which needs to be moved from the back to the front of the environment.

lem, each of which requires a different r-parameter value. There is an improvement of 11% in this example using the current implementation of the kd-tree approach.

The final experiment involves two PUMA robots holding an object, which needs to be moved from the back to the front of the environment (Figure 11). The kd-tree approach gives an 8 times improvement in the running time of the RRT algorithm.

Based on our experiments we have observed that the improvement is significant, when the kd-tree-based dynamic-domain effectively captures the intrinsic dimensionality and shape of the valid configuration space. There is a relationship between the complexity of the space, and the complexity of the regions that can be described by the collection of boxes represented by the dynamic-domain. For example, in Figure 10, the obstacles create multiple holes of various shapes in the valid configuration space, which is hard for the kd-tree to capture. Whereas, in Figure 11 the valid configuration space of similar dimension is significantly less complex, that allows the kd-tree-based approach effectively take advantage of.

Besides the experiments reported in this paper, we conducted around twenty other experiments for both basic motion planning problems, and problems involving closed kinematic chains. We observed that the approach using kd-trees provides significant improvement over the original RRT method on most of the problems we have tested. The improvement is either by orders of magnitudes, or just by several times. Only for three out of twenty problems we noticed either no significant improvement, or a slight deterioration (not more than 10%) in the performance of the kd-tree-based algorithm. This suggests that our approach is promising to provide uniform running time improvement on a large set of motion planning problems.

6 Conclusions and Future Work

We presented a general method for solving motion planning problems which involves constrained feasible configuration spaces. The approach builds a kd-tree representation of the explored part of the configuration space, which enables the RRT to use local information to rapidly explore on the feasible space. The experimental results suggest that it gives uniform improvement over large class of motion planning problems. The algorithm did not provide a running time improvement only on few out of a dozen experiments.

Our next development addresses the implementation of the adaptive tuning of the parameter used in our method. This can be done using the information history from the collision detector. Another important research direction is to address systems with differential constraints. The expression for kinematic closure constraints is similar to the differential constraints, which suggests that our approach may be beneficial for a broader class of motion planning problems.

Kd-trees provide good performance on up to 50-dimensional problems. For problems involving higher dimensions, other techniques need to be developed. An obvious future direction is to use a dimensionality reduction technique, which would project all of the space on significant dimensions, after which the kd-tree approach is applied. This would address the problem of motion planning for thousands of links.

Acknowledgements This work is partially supported by Toyota Future Project Division Grant, NSF CISE-0535007. We are grateful to Benjamin Tovar, Steve Lindemann, and Sariel Har-Peled for valuable discussions. We thank Frank Lingelbach for providing the model of the PUMA robot.

References

1. S. Arya and D. M. Mount. Approximate nearest neihgbor queries in fixed dimensions. In *ACM-SIAM Sympos. Discrete Algorithms*, pages 271–280, 1993.
2. A. Atramentov and S. M. LaValle. Efficient nearest neighbor searching for motion planning. In *IEEE Int'l Conf. on Robotics and Automation*, pages 632–637, 2002.

3. O. B. Bayazit, D. Xie, and N. M. Amato. Iterative relaxation of constraints: A framework for improving automated motion planning,. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, pages 586 – 593, 2005.
4. J. F. Canny. *The Complexity of Robot Motion Planning*. MIT Press, Cambridge, MA, 1988.
5. J. Cortés and T. Siméon. Sampling-based motion planning under kinematic loop-closure constraints. In *6th International Workshop on Algorithmic Foundations of Robotics*, pages 59–74, 2004.
6. A. Dickenstein and I. Z. Emiris. *Solving Polynomial Equations: Foundations, Algorithms, and Applications*. Springer, 2005.
7. E. Ferré and J.-P. Laumond. An iterative diffusion method for part disassembly. In *IEEE Int. Conf. Robot. & Autom.*, 2004.
8. L. Han and N. M. Amato. A kinematics-based probabilistic roadmap method for closed kinematic chains. In *Proceedings of the Workshop on Algorithmic Foundations of Robotics*, 2000.
9. L. Han, L. Rudolph, J. Blumenthal, and I. Valodzin. Stratified deformation space and path planning for a planar closed chain with revolute joints. In *Proceedings of the Workshop on Algorithmic Foundations of Robotics*, 2006.
10. R. S. Hartenburg and J. Denavit. A kinematic notation for lower pair mechanisms based on matrices. *J. Applied Mechanics*, 77:215–221, 1955.
11. L. Jaillet, A. Yershova, S. M. LaValle, and T. Simon. Adaptive tuning of the sampling domain for dynamic-domain RRTs. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, 2005.
12. J. J. Kuffner and S. M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 995–1001, 2000.
13. G. Liu, J. Trinkle, and R. Milgram. Toward complete path planning for planar 3r-manipulators among point obstacles. In *Proceedings of the Workshop on Algorithmic Foundations of Robotics*, 2005.
14. D. M. Mount. ANN programming manual. Technical report, Dept. of Computer Science, U. of Maryland, 1998.
15. J. Porta, L. Ros, and F. Thomas. Multi-loop position analysis via iterated linear programming. In *Proceedings of Robotics: Science and Systems*, Cambridge, USA, June 2006.
16. J. Yakey, S. M. LaValle, and L. E. Kavraki. Randomized path planning for linkages with closed kinematic chains. *IEEE Transactions on Robotics and Automation*, 17(6):951–958, Dec. 2001.
17. A. Yershova, L. Jaillet, T. Simon, and S. M. LaValle. Dynamic-domain RRTs: Efficient exploration by controlling the sampling domain. In *IEEE Int. Conf. Robot. & Autom.*, Barcelona, Spain, 2005.