Asymptotically-optimal Path Planning for Manipulation using Incremental Sampling-based Algorithms

Alejandro Perez Sertac Karaman Alexander Shkolnik Emilio Frazzoli Seth Teller Matthew R. Walter

***Abstract*— A desirable property of path planning for robotic manipulation is the ability to identify solutions in a sufficiently short amount of time to be usable. This is particularly challenging for the manipulation problem due to the need to plan over high-dimensional configuration spaces and to perform computationally expensive collision checking procedures. Consequently, existing planners take steps to achieve desired solution times at the cost of low quality solutions. This paper presents a planning algorithm that overcomes these difficulties by augmenting the asymptotically-optimal RRT* with a sparse sampling procedure. With the addition of a collision checking procedure that leverages memoization, this approach has the benefit that it quickly identifies low-cost feasible trajectories and takes advantage of subsequent computation time to refine the solution towards an optimal one. We evaluate the algorithm through a series of Monte Carlo simulations of seven, twelve, and fourteen degree of freedom manipulation planning problems in a realistic simulation environment. The results indicate that the proposed approach provides significant improvements in the quality of both the initial solution and the final path, while incurring almost no computational overhead compared to the RRT algorithm. We conclude with a demonstration of our algorithm for single-arm and dual-arm planning on Willow Garage's PR2 robot.**

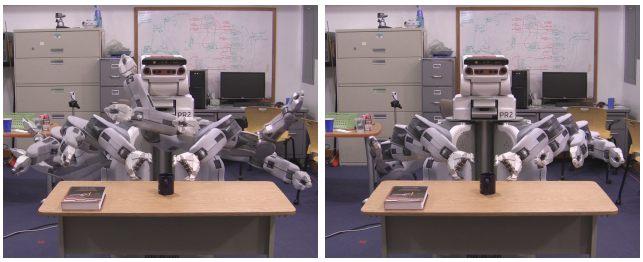
I. Introduction

Motion planning algorithms intended for manipulation tasks must quickly provide plans that not only obey the constraints embedded in the environment, but also take advantage of the full capabilities of the platform in order for the robot to perform challenging tasks in cluttered environments. Along with the requirement of generating paths that leverage the full maneuverability of the platform in some optimal sense, at least two other major challenges play a fundamental role in most manipulation planning tasks.

Firstly, the configuration space of most manipulation platforms is inherently high-dimensional. Robotic arms are typically equipped with joints that result in as many as ten degrees of freedom, making algorithms based on a naive *a priori* discretization of the configuration space impractical. Secondly, as close proximity to certain obstacles is fundamental for manipulation, e.g., for the purpose of grasping, the planning process requires high-precision collision checking of potential trajectories. Often achieved by creating a fine mesh of the manipulator and its surroundings, these collision-checking procedures incur significant computational costs.

Alejandro Perez, Alexander Shkolnik, Seth Teller, and Matthew R. Walter are with the Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA {aperez, shkolnik, teller, mwalter}@csail.mit.edu

Sertac Karaman and Emilio Frazzoli are with the Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA, USA {sertac, frazzoli}@mit.edu



(a) RRT (b) BT+RRT*

Fig. 1. Given the goal of taking both arms of the PR2 from an initial pose underneath the table to the pre-grasp pose with the end effectors near the mug, (a) the RRT typically results a plan involving unnecessary actuation of several joints while (b) our method identifies more efficient plans.

In light of the first challenge, sampling-based algorithms have been shown to generate solutions swiftly in high-dimensional configuration spaces [1]. Arguably, one of the most widely-used algorithms of this class is the Rapidly-exploring Random Tree (RRT) algorithm [2], which grows a search tree starting from the initial configuration by adjoining states sampled randomly from the configuration space.

Most sampling-based algorithms are *probabilistically complete*, i.e., the probability that the algorithm finds a solution, if one exists, converges to one as the number of samples approaches infinity [1], [2]. Indeed, the probability that the algorithm fails to find a solution, when one exists, typically decays to zero very rapidly [2], [3]. The RRT algorithm exhibits this property, efficiently finding an initial feasible solution. However, the algorithm often provides solutions that result in maneuvers that unnecessarily actuate a large number of the platform's joints, paths that traverse round-about trajectories to reach nearby goal configurations, and solutions that require jerky movements. To alleviate this problem, it is common for the solutions returned by the RRT algorithm to be post-processed, e.g., using smoothing algorithms [4]. Even though these methods usually improve the quality of the path considerably, the improvements tend to be local and such algorithms fail to guarantee that the resulting path is an optimal solution in any sense.

Recently, Karaman and Frazzoli [5] have shown that the probability that the RRT algorithm converges to an optimal solution is zero. In the same paper, they propose an alternative method, the RRT*, an incremental sampling-based algorithm with the *asymptotic optimality* property, i.e., almost-sure convergence to an optimal solution. Moreover, it was shown that the RRT* algorithm achieves the asymptotic optimality absent from the RRT, without incurring

a substantial computational overhead. These properties of RRT* provide substantial benefits for manipulation planning. Namely, they enable the planner to quickly find a feasible motion plan and to take advantage of remaining computation time to improve the plan, monotonically converging to the global optimum.

In this paper, we leverage the asymptotic optimality property of the RRT* algorithm to provide close-to-optimal solutions for path planning for manipulation platforms with high-dimensional configuration spaces. We present an implementation of the RRT* that significantly reduces the number of collision checks and, therefore, the computational effort, while still considering numerous paths at every iteration, assuring almost-sure convergence to an optimal solution. We extend this implementation with the Ball Tree algorithm [6] and a memoization technique to improve the performance of the algorithm. Our Monte Carlo evaluations and experimental results on Willow Garage's PR2 platform show that our algorithm is able to plan in high-dimensional configuration spaces, providing significant improvements in the quality of the path without incurring substantial computational overhead, when compared to the RRT.

II. Related Work

The robot motion planning problem has been widely studied for at least three decades [7]. Although the problem has been shown to be computationally challenging [8], several practical approaches have been proposed. Most recently, algorithms that take the quality of the solution into account have received significant attention.

A popular approach is to apply a variant of optimal graph search algorithms like A* to a discretization of the configuration space that is generated offline [9]. Such algorithms have been successfully implemented on robotic cars [10], and very recently applied to manipulation planning problems involving a six-dimensional configuration space [11]. These algorithms, however, are complete and optimal only with respect to the resolution of this discretization. Moreover, the number of discretization points scales exponentially with the number of degrees of freedom, making them impractical for manipulation platforms with a large number of joints.

A more recent line of research has been the investigation of efficiently generating smooth trajectories using, for instance, gradient descent [4] and stochastic optimization [12], which have been applied to solving planning problems for manipulation tasks involving up to seven-dimensional configuration spaces. However, these algorithms are optimal only locally, and are designed for a special class of cost functions that only considers smoothness.

The approach proposed in this paper is globally optimal for a wide class of cost functions, and leverages the efficiency of sampling-based algorithms in high-dimensional configuration spaces. It also has an anytime flavor in the sense that the proposed algorithm provides a feasible solution quickly, and monotonically improves the solution towards an optimal one in the remaining computation time.

III. Problem Definition and Algorithms

A. Problem Definition

Let $X \subseteq \mathbb{R}^d$, referred to as the *configuration space*, be a compact set. The elements of X are called *configurations*. Let $X_{\rm obs}, X_{\rm goal} \subset X$ be open sets, called the *obstacle region* and the *goal region*, respectively. The set defined as $X_{\rm free} := X \setminus X_{\rm obs}$ is called the *obstacle-free space*. A *path* in X is a continuous function $\sigma : [0,1] \to X$. The path σ is said to be *collision-free*, if $\sigma(\tau) \in X_{\rm free}$ for all $\tau \in [0,1]$. The set of all collision-free paths is denoted by $\Sigma_{\rm free}$.

Given an initial configuration $x_{\rm init}$, an obstacle region $X_{\rm obs}$, and a goal region $X_{\rm goal}$, the *motion planning problem* is to find a collision-free path $\sigma : [0,1] \to X_{\rm free}$ that starts from the initial configuration $\sigma(0) = x_{\rm init}$ and reaches the goal region $\sigma(1) \in X_{\rm goal}$.

Let $c : \Sigma_{\rm free} \to \mathbb{R}_{\geq 0}$ be a *cost functional* that maps each collision-free trajectory to a non-negative cost. The *optimal motion planning problem* is to find a collision-free path $\sigma^* : [0,1] \to X_{\rm free}$ that solves the motion planning problem, and moreover minimizes the cost functional $c(\cdot)$, i.e., $c(\sigma^*) = \inf_{\sigma' \in \Sigma_{\rm free}} c(\sigma')$.

B. RRT algorithm

The RRT algorithm was proposed by LaValle and Kuffner [2] as an incremental sampling-based motion planning algorithm. In this section, we first describe some primitive procedures that govern the RRT, and then provide the RRT algorithm in our notation.

Sampling: The `Sample` procedure returns independent uniformly distributed samples from the obstacle-free space.

Collision Checking: Given a path $\sigma : [0,1] \to X$, the `CollisionFree`(σ) procedure returns true iff σ is collision-free, i.e., $\sigma(\tau) \in X_{\rm free}$ for all $\tau \in [0,1]$.

Steering: Given two configurations $x, x' \in X$, the `Steer`(x, x') procedure returns a path $\sigma : [0,1] \to X$ that connects x and x', i.e., $\sigma(0) = x$ and $\sigma(1) = x'$. The `Steer` procedure used in this paper does so with a straight path, i.e., $\sigma(\tau) = (1-\tau)\, x + \tau\, x'$ for all $\tau \in [0,1]$.

Nearest Vertex: Given a set $V \subset X$ of configurations and a configuration $x \in X$, the `Nearest`(V, x) procedure returns the configuration in V that is closest to x with respect to the Euclidean norm, i.e., $\operatorname{argmin}_{x' \in V} \|x' - x\|$.

Finally, in the description of all algorithms to follow, each set A is equipped with `add` and `remove`, which add and remove elements from A, i.e., A.`add`(a) corresponds to setting A to $A \cup \{a\}$.

Algorithm 1 presents the RRT algorithm. The algorithm maintains a tree, denoted as $T = (V, E)$, where $V \subset X$ and $E \subset V \times V$ are called the sets of vertices and edges, respectively. Initially, the set vertices of vertices includes only the initial configuration $x_{\rm init}$ and the set of edges is empty (Line 1). In each iteration (Lines 2-8), the algorithm samples a new configuration $x_{\rm new}$ from $X_{\rm free}$ (Line 4), computes the vertex $x_{\rm nearest}$ that is closest to $x_{\rm new}$ (Line 4), and generates a path σ that connects $x_{\rm nearest}$ and $x_{\rm new}$ (Line 5). If this path is collision-free (Line 6), then the new vertex is added to the tree T as a child of $x_{\rm near}$ (Lines 7-8).

Algorithm 1: The RRT Algorithm

```
1  V ← {x_init}; E ← ∅; T ← (V, E);
2  for i = 1 to N do
3      x_new ← Sample(i);
4      x_nearest ← Nearest(V, x_new);
5      σ ← Steer(x_new, x_nearest);
6      if CollisionFree(σ) then
7          V.add(x_new);
8          E.add( (x_nearest, x_new) );
9  return T = (V, E).
```

C. RRT* Algorithm

The RRT*, first introduced by Karaman and Frazzoli [5], is an incremental sampling-based motion planning algorithm that provides an asymptotic optimality guarantee, i.e., almost-sure convergence to optimal solutions, which the RRT algorithm lacks, without incurring substantial computational overhead. In this section, we present the RRT* algorithm along with a set of modifications tailored to reduce the number of calls to the `CollisionFree` procedure, after introducing some extra primitive operations employed by the RRT* algorithm.

Near Vertices: Given a finite set $V \subset X$ of configurations and a configuration $x \in X$, roughly speaking, the $\texttt{Near}(V, x)$ procedure returns the set of all configurations in V that are close to x, where we define closeness as follows. Letting $n := |V|$ be the number of configurations in V, we define $\texttt{Near}(V, x) := \left\{x' \in V \,:\, \|x' - x\| \leq \gamma \left((\log n)/n\right)^{1/d}\right\}$, where γ is a constant independent of n [5]. In other words, $\texttt{Near}(V, x)$ is the set of all configurations in V that lie inside a ball of volume $O((\log n)/n)$ centered at x.

Lists and Sorting: A list L is an ordered set of elements. Just like sets, each list L is equipped with the $L.\texttt{add}(a)$ method. We will also consider lists of cost, configuration, and path triplets, i.e., triplets of the form (c_i, x_i, σ_i), where $c_i \in \mathbb{R}_{\geq 0}$, $x_i \in X$, and $\sigma_i \in \Sigma_{\text{free}}$. Given a list L of such pairs, the $L.\texttt{sort}()$ method sorts the elements of L according to their cost in the ascending order. When the algorithms iterate through the list's elements, they do so respecting the list ordering.

Cost Functional: Given a vertex x of the tree maintained by the RRT* algorithm, we let $\texttt{Cost}(x)$ be the cost of the unique path that starts from the root vertex x_{init} and reaches x along the vertices of the tree. With a slight abuse of notation, we denote the cost $c(\sigma)$ of a path $\sigma : [0, 1] \to X$ as $\texttt{Cost}(\sigma)$ for notational simplicity.

Algorithm 2 outlines the implementation of the the RRT* algorithm, specifically tailored to reduce the number of calls to the `CollisionFree` procedure. In what follows, we present the algorithm.

Similar to the RRT algorithm, the RRT* iteratively maintains a tree structure, with four key phases.

In the first phase, the RRT* algorithm samples a new configuration x_{new} from X_{free} (Line 3), and computes the set X_{near} of all vertices that are close to x_{new} (Line 4). If X_{near} is an empty set, then X_{near} is updated to include the vertex in the tree that is closest to x_{new} (Lines 5-6).

Algorithm 2: The RRT* Algorithm

```
1  V ← {x_init}; E ← ∅; T ← (V, E);
2  for i = 1 to N do
3      x_new ← Sample(i);
4      X_near ← Near(V, x_new);
5      if X_near = ∅ then
6          X_near ← Nearest(V, x_new);
7      L_near ← PopulateSortedList(X_near, x_near);
8      x_parent ← FindBestParent(L_near, x_new);
9      if x_parent ≠ NULL then
10         V.add(x_new);
11         E.add( (x_parent, x_new) );
12         E ← RewireVertices(E, X_near, x_new);
13 return T = (V, E).
```

Algorithm 3: $\texttt{PopulateSortedList}(X_{\text{near}}, x_{\text{new}})$

```
1  L_near ← ∅;
2  for x_near ∈ X_near do
3      σ_near ← Steer(x_near, x_new);
4      c_near ← Cost(x_near) + Cost(σ);
5      L_near.add( (c_near, x_near, σ_near) );
6  L_near.sort();
7  return L_near;
```

Algorithm 4: $\texttt{FindBestParent}(L_{\text{near}}, x_{\text{new}})$

```
1  for (c_near, x_near, σ_near) ∈ L do
2      if CollisionFree(σ_near) then
3          return x_near;
4  return NULL
```

Algorithm 5: $\texttt{RewireVertices}(E, L_{\text{near}}, x_{\text{new}})$

```
1  for (c_near, x_near, σ_near) ∈ L do
2      if Cost(x_new) + c(σ_near) < Cost(x_near) then
3          if CollisionFree(σ_near) then
4              x_oldparent ← Parent(E, x_near);
5              E.remove( (x_oldparent, x_near) );
6              E.add( (x_new, x_near) );
7  return E
```

In the second phase, the algorithm calls the $\texttt{PopulateSortedList}(X_{\text{near}}, x_{\text{near}})$ procedure (Line 7). This procedure, given in Algorithm 3, returns a sorted triplets of the from $(c_{\text{near}}, x_{\text{near}}, \sigma_{\text{near}})$, for all $x_{\text{near}} \in X_{\text{near}}$, where (i) σ_{near} is the straight path that connects x_{near} and x_{new} and (ii) c_{near} is the cost of reaching x_{new} by following the unique path in the tree that reaches x_{near} and then following σ_{near} (see Line 4 of Algorithm 3). The triplets of the returned list are sorted according ascending cost. Note that at this stage, the paths σ_{near} are *not* guaranteed to be collision-free.

In the third phase, the RRT* algorithm calls the `FindBestParent` procedure, given in Algorithm 4, to determine the minimum-cost collision-free path that reaches x_{new} through one of the vertices in X_{near}. With the vertices

presented in the order of increasing cost (to reach x_{near}), Algorithm 4 iterates over this list and returns the first vertex x_{near} that can be connected to x_{new} with a collision-free path. If no such vertex is found, the algorithm returns NULL.

If the FindBestParent procedure returns a non-NULL vertex x_{parent}, the final phase of the algorithm inserts x_{new} into the tree as a child of x_{parent}, and calls the RewireVertices procedure to perform the "rewiring" step of the RRT* [5]. In this step, the RewireVertices procedure, given in Algorithm 5, iterates over the list L_{near} of triplets of the form $(c_{\text{near}}, x_{\text{near}}, \sigma_{\text{near}})$. If the cost of the unique path that reaches x_{near} along the vertices of the tree is higher than reaching it through the new node x_{new}, then x_{new} is assigned as the new parent of x_{near}.

This implementation of the RRT* is only slightly different than that presented in [5], preserving both computational efficiency and the asymptotic optimality. This implementation is specifically tailored for cases when the collision checking procedure is computationally expensive. Our implementation avoids calling this procedure several time in two places. Firstly, inside the FindBestParent procedure, the CollisionFree procedure is called until a vertex x_{near} that can be connected to x_{new} with a collision-free path is found. The authors have empirically noticed that such a vertex is most often found quickly without iterating through the whole list. Secondly, in the RewireVertices procedure, the CollisionFree procedure is called only if the cost of the resulting path improves the cost to reach to a particular vertex $x_{\text{near}} \in X_{\text{near}}$.

D. Ball Tree Algorithm

The Ball Tree algorithm, presented by Shkolnik and Tedrake [6], is a sampling-based method similar to the RRT that approximates connected regions of free space with balls instead of points. Treated as sets of reachable points, the algorithm uses these balls to perform rejection sampling, resulting in trees that are sparser than those of the standard RRT while maintaining probabilistic completeness.

Algorithm 6 outlines the "inexact" version of the Ball Tree algorithm. A tree is grown in a similar manner as the RRT. Each node in the Ball Tree consists of a ball in configuration space, and is parameterized by the location of the center of the ball and its radius. The ball approximates a portion of configuration space that is reachable from the center of the ball, as the algorithm makes the implicit assumption that any point within a ball is reachable from the ball's center, until proven otherwise. An edge in the tree corresponds to a feasible (verified collision-free) action from the center of one ball to the center of the next ball.

When a node is added to the tree, the algorithm initializes the radius of it's corresponding ball to r_0. In our implementation, r_0 is ∞. The method performs rejection sampling to find a point in configuration space that is not within any of the balls in the tree. Rejected samples are collision checked. If a collision is found, the radius of the nearest enclosing ball is reduced to the distance between the sample and the center of the ball. Note that verifying a single point for collision is relatively inexpensive.

Algorithm 6: The Ball Tree Algorithm

```
1  V ← {x_init, r = 0}; E ← ∅; T ← (V, E);
2  for i = 1 to N do
3      while true do
4          x_new ← Sample(i);
5          if InsideBall(x_new, T) then
6              if !CollisionFree(x_new) then
7                  x_nearest ← NearestBall(V, x_new);
8                  TrimRadius(x_nearest, ||x_new − x_nearest||);
9          else
10             break;
11     x_nearest ← NearestBall(V, x_new);
12     σ ← Steer(x_new, x_nearest);
13     if CollisionFree(σ) then
14         V.add(x_new, r_0);
15         E.add( (x_nearest, x_new) );
16     else
17         TrimRadius(x_nearest, ||x_new − x_nearest||);
18 return T = (V, E).
```

When a sample is found to lie outside all balls in the tree, the Steer procedure extends the tree towards the accepted sample. If a collision-free path is found, a ball centered around the sample point is added to the tree. On the other hand, if a collision is encountered, the radius of the parent node, x_{nearest}, is trimmed to be equal to the length of the collision-free portion of the path.

In this manner, the Ball Tree algorithm fills easy-to-reach neighborhoods with balls. Using rejection sampling, the algorithm focuses its attention on expanding towards unexplored (or previously unattainable) regions of configuration space.

E. Memoized Collision Checking

Memoization [13] is a technique commonly used to avoid redundant function calls by recording results of previous queries. We implement this technique in the CollisionFree procedure to alleviate the computational burden typically associated with checking paths for collisions.

The MemoizedCollisionFree(x) procedure estimates the collision status of a configuration x. To do so, the procedure maintains and queries a cache of collision checks that takes the form of a hash table. The HashIndex(x) function generates the configuration state's index within this table. In the case of a manipulator, this function incorporates an array of resolution values for each joint to discretize its range of motion. Higher resolutions are assigned to joints with movements that result in significant changes to the position of the end effector. When queried, MemoizedCollisionFree(x) uses the index returned by HashIndex(x) to search for the matching cell in the hash table. If the cell is populated, the stored value is returned. If no match exists, the procedure calls CollisionFree(x) and stores the result.

This procedure results in a non-conservative approximation of a precise collision checker and is used as an admissible heuristic. In this paper we verify the final solutions with the regular CollisionFree procedure before execution.

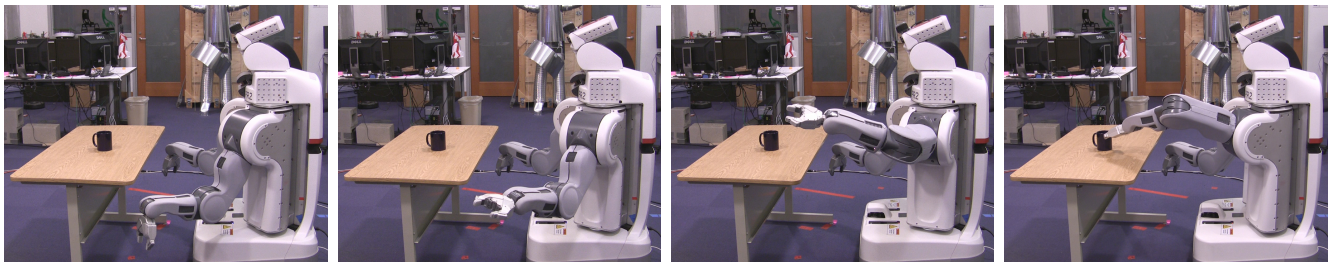


Fig. 2. The planner is tasked with finding a collision-free solution from (left) an initial pose in which the arm is under the table to (right) the goal pose.

F. BT+RRT*: RRT* with Ball Trees and Memoization

We propose a manipulation planning algorithm that offers two compelling advantages. Firstly, it is noticeably faster than conventional planners at identifying an initial, low-cost, feasible path to the goal in configuration space. Secondly, the algorithm is uniquely able to take advantage of available computation time to refine this solution towards an optimal one. We achieve these characteristics by combining the Ball Tree algorithm, which maintains sparse trees to efficiently reach the goal, with the RRT* algorithm, presented in Section III-C, which provides the anytime refinement of the tree. The memoized collision checking procedure, meanwhile, enables efficient implementation of the Ball Tree sampling and steering functions.

More precisely, the algorithm operates as described in Algorithm 2 subject to the following modifications. Firstly, each call to the `CollisionFree` procedure is replaced with the `MemoizedCollisionFree` process. Among them are those calls within the RRT* algorithm as well as the Ball Tree, including the collision checking that is performed as part of the rejection sampling for the "inexact" version of the algorithm. Secondly, until the first feasible solution is found, the sampling step (Line 3) is replaced with Lines 2-9 of the Ball Tree algorithm (Algorithm 6) and the `TrimRadius` procedure (Line 16 of Algorithm 6) is executed whenever the `MemoizedCollisionFree`(σ) procedure fails, i.e., the path σ is not collision-free.

In the next section, this approach is evaluated in Monte-Carlo simulation experiments and compared with other approaches, including both the RRT and the RRT* algorithms.

IV. Results

In this section, we evaluate the effectiveness of our algorithm through both simulation as well as through experiments on the PR2 robot. We first perform a Monte Carlo study to analyze the algorithm's performance on two different planning problems for the PR2 robot. The first involves finding an collision-free path through configuration space that brings a single, seven degree of freedom arm to a pre-grasp pose. In the second scenario, we consider jointly planning trajectories for both arms. The experiments were performed in the OpenRAVE simulation environment [14].

A. Single-Arm Scenario (Seven Degrees of Freedom)

In the first scenario, we simulate an environment in which the PR2 is at rest with its left arm underneath a table as depicted in Figure 2. The task is to plan the trajectories of the seven joints that bring the left arm from this initial pose to a pre-grasp pose above the table. The primary challenge is to negotiate the narrow region of configuration space that is induced by the close proximity of the table.

We applied our planning method to this problem along with the RRT, the RRT*, and a variant of our algorithm in which rewiring takes place only after an initial solution is found (referred to as the BT/RRT*). Each planner performed memoized collision checks and ran for a total of 4000 iterations. We performed a total of 100 Monte Carlo simulations for each algorithm.

TABLE I

Seven Degree of Freedom Monte Carlo Results

		BT+RRT*	RRT	RRT*	BT/RRT*
Success Rate (100 runs)		100.00%	87.00%	99.00%	100.00%
First Solution	Time (s)	2.52 (3.07)	9.75 (12.52)	7.92 (10.97)	2.51 (2.48)
	Cost	7.61 (2.11)	14.73 (5.49)	8.11 (1.67)	17.99 (5.63)
Final Solution	Time (s)	77.14 (4.49)	54.96 (4.75)	77.85 (3.95)	79.21 (4.47)
	Cost	5.52 (0.53)	14.73 (5.49)	5.65 (0.50)	5.67 (0.51)
Time per Iteration (ms)		19.33 (1.13)	13.78 (1.19)	19.51 (0.99)	19.85 (1.12)

Table I summarizes the performance of the different motion planning algorithms. All four planners find a feasible solution in this seven-dimensional configuration space in the majority of the runs. Our method is successful for all of the 100 simulations as is the other planner that utilizes the Ball Tree algorithm. The RRT* is successful in all but one run while the standard RRT planner succeeds 87% of the time. As we discuss later, the improved performance of the RRT* is a consequence of our variation whereby we connect a sample with the lowest cost node in the tree that is collision free. This has the effect of increasing the number of nodes in the tree, improving the likelihood of finding a solution.

The algorithms differ more significantly with regards to the time required to find a solution and its corresponding cost. The RRT and RRT* required a similar amount of time to find an initial solution, with the RRT* being slightly faster with an average time of 7.92 seconds as opposed to 9.75 seconds. We attribute this improvement to the memoized collision checking, without which the average time required for the RRT* to find the first solution is 18.96 seconds compared to 14.22 seconds for the RRT. Not surprisingly, however, the RRT* yielded solutions with significantly lower cost (mean of 8.11 radians) than those of the RRT (mean of 14.73 radians). Meanwhile, with an average time of 2.51 seconds, the Ball Tree-only algorithm required far less time to identify

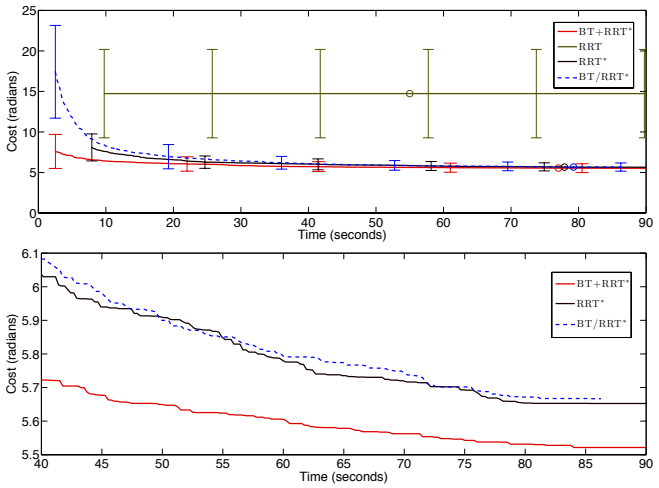


Fig. 3. Solution cost as a function of computation time, averaged over the set of single-arm Monte Carlo simulations for the four planning algorithms. Vertical bars indicate standard deviation over the 100 runs while the open circles denote the average completion time. The bottom figure presents an inset view that compares the mean behavior of the three algorithms that utilize the RRT*.

an initial solution, in exchange for a greater solution cost (mean of 17.99 radians). In contrast, our algorithm found solutions in the same amount of time while also consistently outperforming the other planners in terms of cost. Of the 100 Monte Carlo runs, our method returned solutions with a mean cost of 7.61 radians in an average of only 2.52 seconds. This performance was consistent, as demonstrated by the low variance in the first solution time and the corresponding cost.

Upon finding the an initial solution, our algorithm forgoes the rejection sampling of the Ball Tree and proceeds to build and rewire the tree. In this way, the algorithm attempts to utilize available computation time to improve the quality of the solution. In essence, by effectively switching to an RRT*, one would expect our algorithm and the Ball Tree-only variation to behave similarly to the RRT* given enough iterations. Indeed, this is what we see, as Figure 3 indicates. The plot depicts the solution cost as a function of time averaged over the Monte Carlo simulations for each of the four planners. The vertical bars depict the standard deviation of the cost. Our algorithm exhibits convergence very similar to that of the RRT*, with the advantage of an earlier initial solution time. This is evident in the lower plot in which the two cost profiles are the same, but with that of our algorithm shifted forward in time. The final solution cost after 4000 iterations is nearly identical for these two algorithms as well as the BT/RRT*, as reflected by the values in Table I. While slightly lower, the average cost of the final trajectory for the BT+RRT* is nearly identical to that of the RRT* and the BT/RRT* and exhibits similarly low variance. The RRT, on the other hand, does not refined the initial solution and generates greater average cost.

While our algorithm is faster at finding initial solutions whose cost is similar to the RRT*, this does not come at the expense of increased computational overhead. The average time for each iteration is nearly identical to that of the RRT* and of the BT/RRT* algorithm.

B. Dual-Arm Scenario (Twelve Degrees of Freedom)

Next, we consider jointly planning the motion of both robot arms to achieve a pre-grasp pose. We omit the roll joint of each wrist, resulting in a twelve-dimensional search space. In this scenario, depicted in Figure 4, the robot starts with both arms below the table and is tasked with finding a collision-free trajectory that ends with both grippers positioned to execute a grasping maneuver. We performed 100 simulations of each of the four planning algorithms, each to a total of 6000 iterations. All simulations utilized the memoized collision checking.

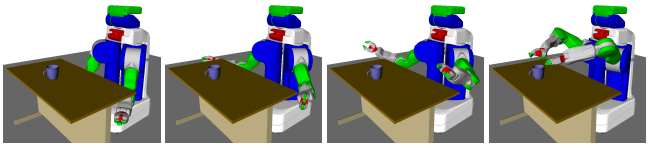


Fig. 4. In the second scenario, the planner is tasked with finding an collision-free trajectory from (left) an initial pose in which both arms are under the table to (right) a pre-grasp goal pose.

TABLE II
TWELVE DEGREE OF FREEDOM MONTE CARLO RESULTS

		BT+RRT*	RRT	RRT*	BT/RRT*
Success Rate (100 runs)		100.00%	58.00%	85.00%	100.00%
First Solution	Time (s)	9.74 (12.84)	29.92 (34.05)	24.61 (32.09)	8.94 (11.06)
	Cost (rad)	8.59 (2.16)	19.76 (5.69)	8.71 (2.34)	22.13 (7.72)
Final Solution	Time (s)	135.28 (15.08)	112.41 (19.46)	131.38 (14.49)	165.28 (28.16)
	Cost (rad)	7.53 (1.21)	19.76 (5.69)	7.97 (1.71)	8.83 (1.73)
Time per Iteration (ms)		22.58 (2.52)	18.77 (3.25)	21.93 (2.42)	27.59 (4.70)

Table II depicts the results of the Monte Carlo simulations. Both algorithms that utilize the Ball Tree were able to find a solution in each run, while the RRT and RRT* succeeded 58% and 85% of the time, respectively. We again attribute the higher success rate for the RRT* to its choice of the best collision-free parent when adding nodes. As with the seven degree of freedom experiments, both the BT/RRT* algorithm and our planner identify initial solutions in significantly less time than the RRT and RRT*. The average cost of the first solution that our algorithm returns, however, is consistently lower and similar to that of the initial RRT* solution.

The algorithm then proceeds to use the available computation time to refine this solution. Figure 5 depicts the average improvement in cost as a function of time along with an indication of the standard deviation. We again see that our algorithm behaves similarly to the RRT* with regards to solution cost. After the maximum number of iterations, our method yields final costs that are slightly lower than the average RRT* cost, with similarly low variance. The results suggest that the same is true of the BT/RRT* planner and we expect that the three algorithms would converge if given a sufficient number of iterations, as in the single-arm scenario.

C. Dual-Arm Scenario (Fourteen Degrees of Freedom)

In the final set of Monte Carlo evaluations, we again consider the dual-arm scenario with the addition of the roll

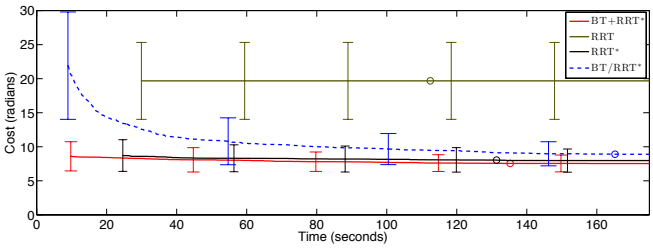


Fig. 5. Solution cost as a function of computation time for the dual-arm planning problem with twelve degrees of freedom. The plots reflect the average over the set of Monte Carlo simulations for the four planning algorithms. Vertical bars indicate standard deviation over the 100 runs while the open circles denote the average completion time.

joint for each wrist, giving a total of fourteen degrees of freedom. We performed 100 simulations of the four planners, allowing each to run for a total of 10,000 iterations.

TABLE III

FOURTEEN DEGREE OF FREEDOM MONTE CARLO RESULTS

		BT+RRT*	RRT	RRT*	BT/RRT*
Success Rate (100 runs)		100.00%	25.00%	59.00%	99.00%
First Solution	Time (s)	34.76 (60.06)	70.78 (82.90)	106.20 (108.65)	23.81 (38.79)
	Cost (rad)	9.82 (2.94)	21.00 (7.69)	10.03 (2.61)	25.46 (9.08)
Final Solution	Time (s)	374.65 (46.46)	263.16 (30.40)	380.82 (34.20)	406.00 (59.20)
	Cost (rad)	8.64 (1.95)	21.00 (7.69)	9.28 (2.14)	10.58 (2.32)
Time per Iteration (ms)		37.50 (4.65)	26.34 (3.04)	38.12 (3.42)	40.64 (5.93)

Table III presents the simulation results. With a maximum of 10,000 iterations, the RRT* was able to find a solution in 59 of the runs and the RRT was successful in only 25. The BT/RRT* planner identified a solution in all but one run while our algorithm found a trajectory every time. Much like the seven and twelve degree of freedom simulations, the BT+RRT* and BT/RRT* Ball Tree planners return an initial solution much sooner than the RRT and RRT*. On average, our algorithm takes longer than the BT/RRT* to isolate an initial solution, though with the benefit of a significant improvement in cost that resembles that returned much later by the RRT*, both in terms of mean cost and variance. After 10,000 iterations, the BT+RRT* yields an average trajectory cost slightly better than that of the RRT* and BT/RRT*.

D. PR2 Experimental Validation

In addition to the Monte Carlo simulations, we utilized our algorithm to execute both the single-arm and dual-arm scenarios on the actual PR2 platform. We demonstrated our planner together with the standard RRT approximately a dozen times for each of the two cases. In the single-arm scenario, both algorithms were permitted 1000 iterations, while the dual-arm application considered 2000 iterations. Figure 1(b) presents a time lapse image that shows the typical trajectories that result from our planner. We compare this with the RRT solutions that typically require excessive arm motion. The consistency with which our algorithm plans efficient paths through configuration space supports the small variance in the lower cost solutions found in the Monte Carlo simulations. Videos that show single-arm and dual-arm planning with our algorithm on the PR2 robot are available at `http://ares.lids.mit.edu/videos/manipulation/`.

V. CONCLUSION

Incremental sampling-based motion planners, such as the RRT, are able to identify feasible motion plans quickly, making them appealing for manipulation. However, the solutions returned by these planners are often far from optimal and the exploration of the space is commonly sacrificed to avoid computationally-expensive collision checking. This paper described a sampling-based planning algorithm that leverages the efficient planning capabilities of the Ball Tree algorithm together with the asymptotic optimality provided by the RRT*. Moreover, the algorithm delays checking paths for collision until it is absolutely necessary and leverages memoization to reduce its computation time. We employed Monte Carlo simulations to evaluate the ability of this algorithm to provide low-cost solutions for high-dimensional planning problems in a timely fashion. We further demonstrated the algorithm's performance through experiments that involved planning single and dual-arm trajectories on the PR2 robot.

ACKNOWLEDGEMENTS

The authors are grateful to Professors L.P. Kaelbling and T. Lozano-Perez and to Willow Garage Inc. for providing access to the PR2 experimental platform.

REFERENCES

[1] L. E. Kavraki, P. Svestka, J. C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Trans. on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, August 1996.

[2] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *Int'l J. of Robotics Research*, vol. 20, no. 5, pp. 378–400, May 2001.

[3] L. E. Kavraki, M. N. Kolountzakis, and J. C. Latombe, "Analysis of probabilistic roadmaps for path planning," *IEEE Trans. on Robotics and Automation*, vol. 14, no. 1, pp. 166–171, February 1998.

[4] N. Ratliff, M. Zucker, J. Bagnell, and S. Srinivasa, "CHOMP: Gradient optimization techniques for efficient motion planning," in *Proc. IEEE Int'l Conf. on Robotics and Automation (ICRA)*, May 2009, pp. 489–494.

[5] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *Int'l J. of Robotics Research*, vol. 30, no. 7, pp. 846–894, June 2011.

[6] A. Shkolnik and R. Tedrake, "Sample-based planning with volumes in configuration space," http://groups.csail.mit.edu/robotics-center/public_papers/Shkolnik11.pdf, 2011 (To be submitted).

[7] T. Lozano-Perez and M. A. Wesley, "An algorithm for planning collision-free paths among polyhedral obstacles," *Comm. ACM*, vol. 22, no. 10, pp. 560–570, October 1979.

[8] J. Canny and J. H. Reif, "New lower bound techniques for robot motion planning problems," in *IEEE Symp. on Foundations of Computer Science (FoCS)*, Los Angeles, CA, October 1987, pp. 49–60.

[9] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun, "Anytime search in dynamic graphs," *Artificial intelligence J.*, vol. 172, no. 14, pp. 1613–1643, September 2008.

[10] M. Likhachev and D. Ferguson, "Planning long dynamically-feasible maneuvers for autonomous vehicles," *Int'l J. of Robotics Research*, vol. 28, no. 8, pp. 933–945, August 2009.

[11] B. Cohen, G. Subramanian, S. Chitta, and M. Likhachev, "Planning for manipulation with adaptive manipulation primitives," in *Proc. IEEE Int'l Conf. on Robotics and Automation (ICRA)*, May 2011.

[12] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal, "STOMP: Stochastic trajectory optimization for motion planning," in *Proc. IEEE Int'l Conf. on Robotics and Automation (ICRA)*, May 2011.

[13] D. Michie, "Memo functions and machine learning," *Nature*, vol. 218, pp. 19–22, April 1968.

[14] R. Diankov, "Automated construction of robotic manipulation programs," Ph.D. dissertation, Carnegie Mellon University, Robotics Institute, August 2010.