Fast probabilistic collision checking for sampling-based motion planning using locality-sensitive hashing

2 authors, including:

Dinesh Manocha
University of Maryland, College Park
811 PUBLICATIONS **22,067** CITATIONS

Some of the authors of this publication are also working on these related projects:

Efficient Multi-Agent Global Navigation Using Interpolating Bridges View project

Crowd Management Intelligent-based Notification System View project

Fast Probabilistic Collision Checking for Sampling-based Motion Planning using Locality-Sensitive Hashing

Jia Pan[1] and Dinesh Manocha[2]

Abstract—We present a novel approach to perform fast probabilistic collision checking in high-dimensional configuration spaces to accelerate the performance of sampling-based motion planning. Our formulation stores the results of prior collision queries, and then uses such information to predict the collision probability for a new configuration sample. In particular, we perform an approximate k-NN (k-nearest neighbor) search to find prior query samples that are closest to the new query configuration. The new query sample's collision status is then estimated according to the collision checking results of these prior query samples, based on the fact that nearby configurations are likely to have the same collision status. We use locality-sensitive hashing techniques with sub-linear time complexity for approximate k-NN queries. We evaluate the benefit of our probabilistic collision checking approach by integrating it with a wide variety of sampling-based motion planners, including PRM, lazyPRM, RRT, and RRT*. Our method can improve these planners in various manners, such as accelerating the local path validation, or computing an efficient order for the graph search on the roadmap. Experiments on a set of benchmarks demonstrate the performance of our method, and we observe up to 2x speedup in the performance of planners on rigid and articulated robots.

I. INTRODUCTION

Motion planning is an important problem in robotics, virtual prototyping and related areas. Most practical methods for motion planning of high-DOF (degrees-of-freedom) robots are based on random sampling in configuration spaces, including PRM [1] and RRT [2]. The resulting algorithms avoid explicit computation of obstacle boundaries in the configuration space ($\mathcal{C}$-space) and instead use sampling techniques to compute paths in the free space ($\mathcal{C}_{\text{free}}$). The main computations include probing the configuration space for collision-free samples, joining nearby collision-free samples by local paths, and checking whether the local paths lie in the free space. There is extensive work on different sampling strategies, faster collision checking, and on biasing the samples to handle narrow passages according to local information.

In motion planning, the collision detection module is typically used as an oracle for collecting information about the free space and approximating its topology. This module classifies a given configuration or a local path as either collision-free (i.e., in $\mathcal{C}_{\text{free}}$) or in-collision (i.e., overlapping with $\mathcal{C}_{\text{obs}}$). Most motion planning algorithms only store the collision-free samples and local paths, and use them to compute a global

J. Pan is with the Department of Computer Science, the University of Hong Kong; D. Manocha is with the department of Computer Science, the University of North Carolina at Chapel Hill.

path from the initial configuration to the goal configuration. Typically, the in-collision configurations or local paths are discarded.

In order to accelerate the performance of sampling-based planners, our goal is to improve the performance of the collision detection module by leveraging the information about prior collision queries. This notion of using the results of previous queries is not new, and has been used for motion planning. For instance, a variety of planners [3], [4], [5], [6] utilize the in-collision configurations or the samples near the boundary of the configuration obstacles ($\mathcal{C}_{\text{obs}}$) to bias the sample generation or to improve the planners' performance in narrow passages. However, it can be expensive to perform geometric inference based on the outcome of a large number of collision queries in high-dimensional configuration spaces. As a result, most prior planners only use partial or local information about configuration spaces.

Main Results: We present a novel probabilistic approach which improves the performance of the collision detection module by utilizing the results from prior collision queries, including both in-collision and collision-free samples. Our formulation leverages the historical information generated using collision queries to compute an approximate representation of $\mathcal{C}$-space as a hash table. Given a new probe or collision query in $\mathcal{C}$-space, we perform efficient inference on the approximate $\mathcal{C}$-space in order to compute a collision probability for this query. This probability is used either as a similarity result or as a prediction of the exact collision query. Based on this collision probability, we design a collision filter for efficient milestone and local path validation, which can greatly improve the performance of sampling-based motion planners.

The underlying prediction performed on the approximate $\mathcal{C}$-space is based on k-NN (k-nearest neighbor) queries. The efficiency of the k-NN computation in high-dimensional configuration spaces is achieved by using locality-sensitive hashing (LSH) algorithms, which have sub-linear complexity. In particular, we present a point-point k-NN query for computing the nearest neighbors of a point configuration, and a line-point k-NN algorithm for finding the nearest neighbors of a line query, which arises in the context of local planning. We derive bounds on the accuracy and time complexity of these LSH-based k-NN algorithms and show that the collision probability computed using these algorithms converges to the exact collision detection as the size of dataset increases.

Our approach is general and can be combined with any sampling-based motion planning algorithm. In particular, we

present improved versions of PRM, lazyPRM, and RRT planning algorithms based on our probabilities collision checking algorithm. Furthermore, it is quite efficient for high-dimensional configuration spaces. We have applied these planners to rigid and articulated robots, and have observed up to 2x speedup. The only additional overhead comes from storing the prior instances in the hash table and performing k-NN queries; these account for only a small fraction of the overall planning time. Finally, the learned approximate $\mathcal{C}$-space can be updated efficiently for moving obstacles and can also be used for motion planning in dynamic environments. This paper is a revised and extended version of our prior work [7].

The rest of the paper is organized as follows. We survey related work in Section II. Section III gives an overview of the probabilistic collision checking framework. We present details of the probabilistic collision checking and analyze its accuracy and complexity in Section IV and Section V. We show the integration of our fast collision checking module with a variety of motion planning algorithms in Section VI and highlight the performance of the modified planners on various benchmarks in Section VII.

II. RELATED WORK AND BACKGROUND

In this section, we first provide an overview about how the collision checking module is used in prior sampling-based planners, with a brief comparison with our approach. Next, we discuss different ways adopted by previous sampling-based planners to leverage information accumulated during the planning process about the surrounding environment, and compare these methods with our approximate collision checking module. Finally, we briefly survey the k-nearest neighbor search algorithms, especially the locality-sensitive hashing approaches, which make up our toolbox for accelerating collision queries.

A. Collision Checking for Motion Planning

One important feature of sampling-based motion planners is the use of exact collision queries to probe the connectivity of $\mathcal{C}_{\text{free}}$. However, the topology of $\mathcal{C}_{\text{free}}$ can be rather complex, and may consist of multiple components or small, narrow passages. As a result, it is challenging to capture the full connectivity of $\mathcal{C}_{\text{free}}$ using collision queries. There is extensive work on various techniques improving the connectivity computation with different sampling strategies.

Many sampling approaches used for sampling-based planners tend to be memoryless, i.e., the sampling technique used to generate the $(n + 1)$th sample is independent of the previous n samples. Approaches belong to this category include OBPRM [8], Gaussian sampling [3], retraction-based planners [9], [5], [10], and methods specially designed for narrow passages [6], [1]. All these sampling strategies are orthogonal to our probabilistic collision query approach, and thus our approach can be combined with all these techniques for a better performance of motion planners.

In some recent approaches, adaptive sampling strategies have been proposed that evolve while more information about $\mathcal{C}$-space and $\mathcal{C}_{\text{free}}$ has been inferred via sampling. In other words, these strategies are not memoryless because the underlying approximate representation of $\mathcal{C}$-space changes as more samples are generated. For instance, Jaillet et al. [11] and Yershova et al. [12] approximate the free space using a set of size-varying balls around nodes in the RRT representation. Burns and Brock [13] approximate the $\mathcal{C}$-space with a set of prior samples, either collision-free or in-collision. Recently, Knepper and Mason [14] extend the adaptive sampling approach in [13] to non-holonomic motion planning by defining the utility of local paths. Denny and Amato [4] construct roadmaps in both $\mathcal{C}_{\text{free}}$ and $\mathcal{C}_{\text{obs}}$, for generating more samples in narrow passages.

Our method also computes an approximate representation of $\mathcal{C}$-space, in terms of in-collision and collision-free samples. However, our approach is independent of the underlying sampling strategy, and thus can be combined with all the adaptive sampling strategies mentioned above for better performance. One method directly related with our approach is Burns and Brock [15], which also used k-NN queries to estimate the collision status for a local path based on the database of prior collision queries. There are several important differences between their approach and ours. First, our nearest neighbor queries on a local path uses the line-point k-NN query (see Section IV), which is more accurate and efficient. In particular, we convert this problem into a point-point k-NN problem in a higher dimensional space, and then use LSH technique for efficient query in the higher-dimensional space. Second, a set of initial random samples are used in [15], which are not necessary for our approach. In addition, our approach can also handle dynamic environments with moving obstacles, where we can approximate the underlying representation of $\mathcal{C}$-space. Finally, our approach can be combined with any sampling-based planner, whereas the algorithm proposed by Brock and Burns [15] is mainly for PRMs.

Some of our previous work is also about probabilistic collision checking, such as [16], [17]. However, they mainly focus on the collision checking in environments with noise and uncertainty, and thus are not directly related with this work.

B. Motion Planners and Environment Learning

The performance of motion planners can be improved by exploiting learned knowledge about the underlying geometric structures in tasks and human environments. In particular, this capability is useful for robots working in domestic environments, because these environments do not change much (walls and shelves, for example, are static, and large objects like furniture are not moved frequently). Many approaches have been proposed to help motion planners learn about the surrounding environment by reusing the trajectories planned in the past. For instance, Jetchev and Toussaint [18] construct a database of high-dimensional features which captures information about the proximity of the robot to obstacles. Such information is then used to predict a good path while facing a new situation. Other methods construct a database of past motion plans [19], [20], [21], [22], [23].

The method proposed in this paper enables motion planners to learn about environments from the results of previous

collision detection queries. In our method, a database of collision results is maintained, rather than a database of motion plans. Compared with a database of motion plans, our database of collision results has some advantages. First, it is easier to compute and store this information, and can also be used for a dynamic environment. Second, since the dimension of the motion plan database (i.e., the length of the motion paths) is much larger than that of a database of collision query results (i.e., the dimension of the $\mathcal{C}$-space), the storage and query performance is higher for collision query results than for motion plans. Finally, our method outperforms previous methods [18] that also used k-NN search, due to our improved k-NN computation algorithm.

C. k-Nearest Neighbor (k-NN) Search

The problem of finding the k-nearest neighbor within a database of high-dimensional points is well-studied in various areas, including databases, computer vision, and machine learning. Samet's book [24] provides a good survey of various techniques used to perform the k-NN search. In order to handle large and high-dimensional spaces, most practical algorithms are based on approximate k-NN queries [25]. In these formulations, the algorithm is allowed to return a point whose distance from the query point is at most $1 + \epsilon$ times the distance from the query to its k-nearest points; $\epsilon > 1$ is called the *approximation factor*. One popular approximate k-NN method is the locality-sensitive hashing algorithm, which is originally designed for point k-NN queries, but has also be extended to line queries [26], hyper-plane queries [27] and point/subspace queries [28]. LSH-based k-NN has already been used in motion planning, e.g., a parallel version of LSH-based k-NN was used in a parallel PRM framework [29].

The basic LSH algorithm is an approximate method for computing k-nearest neighbors. The underlying idea is to hash the data items in a *locality-sensitive* manner: similar items are mapped to the same buckets with high probability, and dissimilar items are usually mapped into different buckets, with only a low probability of being sorted into the same buckets. We need only search within a given query's bucket or in nearby buckets to collect the k-nearest neighbors for a given query. As the size of a bucket is much smaller than the number of all possible data items, the search process tends to be more efficient.

In particular, a M-dimensional hash function $g(\cdot)$ is used to divide the entire problem space into a grid and to distribute each data item into one grid cell:

$$g(\mathbf{x}) = [h^1(\mathbf{x}), h^2(\mathbf{x}), ..., h^M(\mathbf{x})], \quad (1)$$

where $h^i(\cdot)$ is a 1-dimensional hash function randomly selected from a hash function family $\mathcal{H}$ with 'locality-sensitiveness', which can be formally described as follows:

Definition [30] Let $h_{\mathcal{H}}$ denote a random choice of hash functions from the function family $\mathcal{H}$, and let $B(\mathbf{x}, r)$ be a radius-r ball centered at $\mathbf{x}$. $\mathcal{H}$ is called $(r, r(1 + \epsilon), p_1, p_2)$-sensitive for $\text{dist}(\cdot, \cdot)$ when for any $\mathbf{x}, \mathbf{x}' \in \mathcal{D}$,
- if $\mathbf{x}' \in B(\mathbf{x}, r)$, then $\mathbb{P}[h_{\mathcal{H}}(\mathbf{x}) = h_{\mathcal{H}}(\mathbf{x}')] \geq p_1$,
- if $\mathbf{x}' \notin B(\mathbf{x}, r(1 + \epsilon))$, then $\mathbb{P}[h_{\mathcal{H}}(\mathbf{x}) = h_{\mathcal{H}}(\mathbf{x}')] \leq p_2$.

For this family of hash functions to be useful, we require $p_1 > p_2$, which indicates that the probability of two points being mapped into the same hash bucket is large while they are close to each other, and is small otherwise.

According to the distance metric used for k-NN search, different hash functions are being used. For instance, the hash function for l_p metric, $p \in (0, 2]$ [31] is $h_i(\mathbf{x}) = \lfloor \frac{\mathbf{a}_i \cdot \mathbf{v} + b_i}{W} \rfloor$, where the vector $\mathbf{a}_i$ consists of i.i.d. entries from Gaussian distribution $N(0, 1)$ and b_i is drawn from a uniform distribution $U[0, W]$. M and W control the dimension and size of each lattice cell, respectively, and therefore control the locality sensitivity of the hash functions. In order to achieve higher accuracy for approximate k-NN queries, L hash tables are used and each of them is constructed independently with different dim-M hash functions $g(\cdot)$. Given a query item $\mathbf{x}'$, we first compute its hash code using $g(\mathbf{x}')$ and locate the hash bucket that contains $\mathbf{x}'$. All the items in the bucket are potential candidates for k-NN computations. Next, we perform a local scan on the candidate set to compute the k-NN results. For the local scan result, the following conclusion holds for the l_2 metric:

Theorem 1: (Point-point k-NN query) [31] Let $\mathcal{H}$ be a family of $(r, r(1 + \epsilon), p_1, p_2)$-sensitive hash functions, with $p_1 > p_2$. Given a dataset of size N, we set $M = \log_{1/p_2} N$ and $L = N^\rho$, where $\rho = \frac{\log p_1}{\log p_2}$. Using L-hash tables over dimension M, given a point query $\mathbf{p}$, with probability at least $\frac{1}{2} - \frac{1}{e}$, the LSH algorithm solves the (r, ϵ)-neighbor problem. In other words, if there exists a point $\mathbf{x}$ that $\mathbf{x} \in B(\mathbf{p}, r(1+\epsilon))$, then the algorithm will return the point with probability $\geq \frac{1}{2} - \frac{1}{e}$. The retrieval time is bounded by $\mathcal{O}(N^\rho)$.
In particular, for the hash function mentioned above, we have $\rho \leq \frac{1}{1+\epsilon}$ and the algorithm has sub-linear complexity, i.e., the results can be retrieved in time $\mathcal{O}(N^{\frac{1}{1+\epsilon}})$.

III. OVERVIEW

In this section, we summarize the notations and symbols used in our paper, and give an overview of our approach using probabilistic collision checking.

A. Notations and Symbols

We denote the configuration space as $\mathcal{C}$-space, and each point within the space represents a configuration $\mathbf{x}$. $\mathcal{C}$-space is composed of two parts: collision-free points ($\mathcal{C}_{\text{free}}$) and in-collision points ($\mathcal{C}_{\text{obs}}$). $\mathcal{C}$-space can be non-Euclidean, but we approximately embed a non-Euclidean space into a higher-dimensional Euclidean space using the Linial-London-Robinovich embed [32] and then perform k-NN queries. We use $\mathcal{D}$ to denote a set of N configuration points $\mathcal{D} = \{\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_N\}$ along with their exact collision statuses, which is an approximation to the exact $\mathcal{C}$-space.

A *local path* in $\mathcal{C}$-space is a continuous curve that connects two configurations. It is difficult to compute $\mathcal{C}_{\text{obs}}$ or $\mathcal{C}_{\text{free}}$ explicitly; therefore, sampling-based planners use collision checking between the robot and obstacles to probe the $\mathcal{C}$-space implicitly. These planners perform two kinds of queries: the *point query* and the *local path query*. We use the symbol Q to denote either of these queries. When it is necessary to

distinguish point and line queries, we use $\mathbf{p}$ for a point query and l for a line query.

We use an operator $y(\cdot)$ to denote the exact collision status (0 for collision-free and 1 for in-collision). In particular, $y(\mathbf{x})$ is the collision status of a configuration sample $\mathbf{x}$, $y(\mathbf{p})$ is the collision status of a point query p, and $y(l)$ is the collision status of a line l. We usually abbreviate $y(\mathbf{x})$ or $y(\mathbf{p})$ by y. The estimated collision status of a query is computed by a binary-class classifier $c(\cdot)$.

We denote $\mathrm{vec}(\cdot)$ as the vectorization of a given matrix. In particular, $\mathrm{vec}(\mathbf{A})$, the vectorization of an $m \times n$ matrix $\mathbf{A}$, is the $mn \times 1$ column vector which is obtained by stacking the columns of the matrix $\mathbf{A}$ on top of one another:

$$\mathrm{vec}(\mathbf{A}) = [a_{1,1}, ..., a_{m,1}, a_{1,2}, ..., a_{m,2}, ..., a_{1,n}, ..., a_{m,n}]^T,$$

where $a_{i,j}$ represents the (i,j)-th element of matrix $\mathbf{A}$.
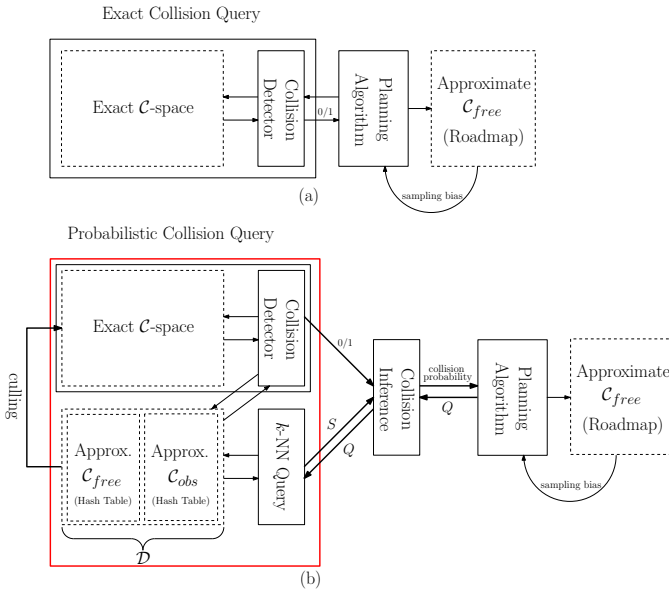

Fig. 1: The collision detection module in sampling-based planners: exact collision checking only (a) and our approach with probabilistic collision checking (b). (a) The collision detection routine is an oracle used by the planner to gather information about $\mathcal{C}_{\mathrm{free}}$ and $\mathcal{C}_{\mathrm{obs}}$. The planner performs binary collision queries, either on point configurations or 1-dimensional local paths, and estimates the connectivity of $\mathcal{C}_{\mathrm{free}}$ (shown as Approximate $\mathcal{C}_{\mathrm{free}}$). Moreover, some planners leverage in-collision results to bias sample generation according to different heuristics. (b) Our method also uses collision queries. However, we store all in-collision results (as Approximate $\mathcal{C}_{\mathrm{obs}}$) and collision-free results (as Approximate $\mathcal{C}_{\mathrm{free}}$). Given a new query, our algorithm first performs a k-NN query on the given configuration or local path and then computes a collision probability for this query. The motion planner then uses the collision probability as a heuristic to guide the exploration process in the configuration space.

B. Probabilistic Collision Checking

Exact collision checking is an important component of sampling-based motion planners. By providing binary collision statuses for configuration points or local paths in the configuration space, collision checking helps the planners to learn about the connectivity of $\mathcal{C}$-space, and eventually to compute a collision-free continuous path connecting the initial and goal configurations in $\mathcal{C}$-space (Figure 1(a)). The collision query results can also bias the planner's sampling scheme thorough different heuristics (e.g., retraction rules).

Unlike the exact collision checking algorithm that computes many collision queries independently, our new probabilistic collision checking scheme exploits the prior collision information accumulated during the planning process, and leverages the spatial correlation between different collision queries (Figure 1(b)). In particular, after the collision checking routine finishes probing the $\mathcal{C}$-space for a given query, we add the obtained information related to this query in a dataset $\mathcal{D}$, which stores all the historical collision query results during the planning process. The stored information is a binary collision status, if the query is a point within $\mathcal{C}$-space, or the collision statuses of several configuration points along the path, if the query is a local path. The resulting dataset $\mathcal{D}$ constitutes the complete set of information we know about $\mathcal{C}$-space, all learned from collision checking routines. Therefore, we use $\mathcal{D}$ as an approximate description of the underlying $\mathcal{C}$-space: $\mathcal{C}_{\mathrm{obs}}$ and $\mathcal{C}_{\mathrm{free}}$ are represented by in-collision samples and collision-free samples, respectively. We then use these samples to estimate the collision status for a new query. The estimation result is in a form of a probability value, i.e., the *collision probability* (refer to Section V for details).
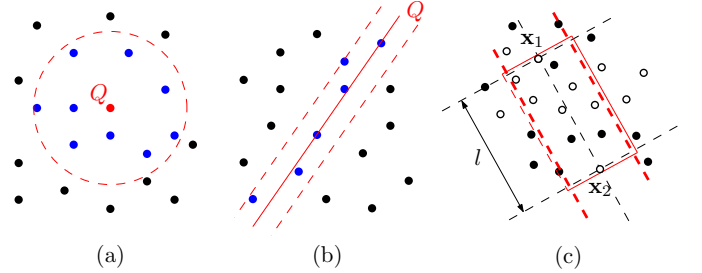

Fig. 2: Two types of k-NN queries used in our method: (a) point-point k-NN; (b) line-point k-NN. Q is the query item, and the results of different queries are shown as blue points in each figure. We present novel LSH-based algorithms for fast computation of these queries. (c) The line-point k-NN query is used to compute prior collision instances that can influence the collision status of a local path connecting $\mathbf{x}_1$ and $\mathbf{x}_2$ in $\mathcal{C}$-space. The query line is the line segment between $\mathbf{x}_1$ and $\mathbf{x}_2$. The white points are prior collision-free samples in the dataset, and the black points are prior in-collision samples.

Given a new query Q, either a point or a local path, we first perform k-NN search on the dataset $\mathcal{D}$ to find its neighbor set S. The set S provides a rough description about the $\mathcal{C}$-space local around the query Q. If S contains sufficient information to infer the collision status of the query, we compute a collision probability for the new query according to S; otherwise, we perform exact collision checking for this query and the query result is added into $\mathcal{D}$. The calculated collision probability provides prior information about a given query's collision status, which is useful in many ways. First, some new query configurations or local paths have a neighborhood which is well-sampled by the database, and thus we can use the collision probability as a culling filter to avoid the exact (and expensive) collision checking for these queries. Second, according to the collision probability, we can

decide an efficient order while performing the exact collision checking for a set of queries. For instance, many planners like RRT need to select a local path that can best improve the local exploration in $\mathcal{C}_{\text{free}}$, i.e., a local path that is both long and collision-free. The collision probability can be used to find an efficient sorting strategy, which thereby reduces the number of exact collision tests.

Two types of k-NN queries involve in our probabilistic collision checking algorithm. One retrieves points closest to a given point query: this is the well-known k-NN query, which we call the *point-point k-NN* query. The second query tries to find the points that are closest to a given line, which arises in the context of local path query. We call this second query the *line-point k-NN* query. These two types of k-NN queries are illustrated in Figure 2. For efficiency, both types of queries are implemented using the locality-sensitive hashing technique. For point-point k-NN queries, we directly build on prior LSH results in Section II-C. For line-point k-NN queries, we will present a new LSH-based algorithm in Section IV. When the collision result for a new configuration query is computed, we calculate the hash code for that query and add it to the hash tables. This operation is performed once for each item stored in the dataset $\mathcal{D}$.

The notion of having sufficient information about S is related to how confident we are about our inferences drawn from S. If the confidence is too small, the algorithm rejects the results of probabilistic collision queries and performs exact collision queries instead. We consider two types of rejection cases: *ambiguity rejection* and *distance rejection* [33]. Ambiguity rejection happens when the collision probability of a given query is nearly 0.5. Distance rejection happens when the query configuration is far (in terms of geometric distance) from all prior instances stored in the database.

An overview of our probabilistic collision framework is given in Algorithm 1.

Algorithm 1: `probabilistic-collision-query(`$\mathcal{D}, Q$`)`

begin
    if Q is point query **then**
        $S \leftarrow$ `point-point-`k`-NN(`Q`)`
        if S provides sufficient information for inference **then**
            `probabilistic-collision-query(`S, Q`)`
        else `exact-collision-query(`$\mathcal{D}, Q$`)` ;
    if Q is line query **then**
        $S \leftarrow$ `line-point-`k`-NN(`Q`)`
        if S provides sufficient information for inference **then**
            `probabilistic-continuous-collision-query(`S, Q`)`
        else `exact-continuous-collision-query(`$\mathcal{D}, Q$`)` ;

IV. LSH-BASED LINE-POINT k-NN QUERY

One of the contributions of this paper is to extend the LSH formulation to the line-point k-NN query, for efficiently estimating the collision status of a local path. In comparison with previous methods for such computations [26], [28], our line-point k-NN results in a more compact form. In addition, we also derive LSH bounds similar to the point-point k-NN, as shown in Theorem 1. Moreover, we address several

issues that arise when using our algorithm for sampling-based motion planning, such as handling non-Euclidean metrics and reducing the dimension of the embedded space.

The simplest algorithm for line-point k-NN query is based on sampling the line into a sequence of uniformly sampled points at a fixed resolution, and using point-point k-NN algorithms on each of those sampled points. One major drawback of such an approach is its efficiency, as we land up performing a high number of point-point k-NN queries for a given line or local path. Furthermore, the samples in the database are typically not distributed in a uniform manner. As a result, it is hard to compute the appropriate sampling resolution for the line.

The main issue in terms of using LSH to perform line-point k-NN query is to embed the line query and the point dataset into a higher-dimensional space, and then to perform point-point k-NN queries in that embedded space. First, we present a technique to perform line-point embedding. Next, we design hash functions for the embedding and prove that these hash functions satisfy the locality-sensitive property for the original data (i.e., $\mathcal{D}$). Finally, we derive the error bound and time bound for the approximate line-point k-NN query, which is similar to that given in Theorem 1.

A. Line-point Distance

A line l in $\mathcal{R}^d$ is described as $l = \{\mathbf{a} + s \cdot \mathbf{v}\}$, where $\mathbf{a}$ is a point in $\mathcal{R}^d$ and $\mathbf{v}$ is a unit vector in $\mathbf{R}^d$. The Euclidean distance of a point $\mathbf{x} \in \mathcal{R}^d$ to the line l is:

$$\text{dist}^2(\mathbf{x}, l) = (\mathbf{x} - \mathbf{a}) \cdot (\mathbf{x} - \mathbf{a}) - ((\mathbf{x} - \mathbf{a}) \cdot \mathbf{v})^2. \quad (2)$$

Given a database $\mathcal{D} = \{\mathbf{x}_1, ..., \mathbf{x}_N\}$ of N points in $\mathcal{R}^d$, the goal of line-point k-NN query is to retrieve the points from $\mathcal{D}$ that are closest to l. We do not directly use Equation 2 for line-point k-NN query, because in that form the database item (i.e., the point) and the query item (i.e., the line) are not well separated. To accelerate the line-point k-NN using LSH-based techniques, we convert the distance metric into a form which is more suitable for efficient k-NN query.

B. Line-point Embedding: Non-affine Case

We first assume the non-affine line query, i.e., l, passes through the origin (i.e., $\mathbf{a} = \mathbf{0}$). In this case, $\text{dist}(\mathbf{x}, l) = \mathbf{x} \cdot \mathbf{x} - (\mathbf{x} \cdot \mathbf{v})^2$, and it can be re-formalized as the inner product of two $(d + 1)^2$-dimensional vectors:

$$
\begin{aligned}
&\text{dist}^2(\mathbf{x}, l) \\
&= \mathbf{x} \cdot \mathbf{x} - (\mathbf{x} \cdot \mathbf{v})^2 \\
&= \text{Tr}(\mathbf{x}^T (\mathbf{I} - \mathbf{v}\mathbf{v}^T)\mathbf{x}) \\
&= \text{Tr}\left(\begin{pmatrix} \mathbf{x} \\ t \end{pmatrix}^T (\mathbf{I} \quad \mathbf{0})^T (\mathbf{I} - \mathbf{v}\mathbf{v}^T) (\mathbf{I} \quad \mathbf{0}) \begin{pmatrix} \mathbf{x} \\ t \end{pmatrix} \right) \\
&= \text{Tr}\left((\mathbf{I} \quad \mathbf{0})^T (\mathbf{I} - \mathbf{v}\mathbf{v}^T) (\mathbf{I} \quad \mathbf{0}) \begin{pmatrix} \mathbf{x} \\ t \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ t \end{pmatrix}^T \right) \quad (3) \\
&= \text{vec}((\mathbf{I} \quad \mathbf{0})^T (\mathbf{I} - \mathbf{v}\mathbf{v}^T) (\mathbf{I} \quad \mathbf{0})) \cdot \text{vec}\left(\begin{pmatrix} \mathbf{x} \\ t \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ t \end{pmatrix}^T \right) \\
&= V(\mathbf{v}) \cdot V(\mathbf{x}),
\end{aligned}
$$

where $\mathbf{I}$ is $d \times d$ identity matrix, $\mathrm{Tr}(\cdot)$ is the trace of a given square matrix and t can be any real value; $\mathrm{vec}(\cdot)$ is the vectorization operation. $V^P(\cdot)$ is an embedding which yields a $(d+1)^2$-dimensional vector from d-dimensional point vector $\mathbf{x}$: $V^P(\mathbf{x}) = \mathrm{vec}(\begin{pmatrix} \mathbf{x} \\ t \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ t \end{pmatrix}^T)$; $V^L(\cdot)$ is an embedding which yields a $(d+1)^2$-dimensional vector from a line $l = \{s\mathbf{v}\}$ in d-dimensional space: $V^L(\mathbf{v}) = \mathrm{vec}(\begin{pmatrix} \mathbf{I} - \mathbf{v}\mathbf{v}^T & \mathbf{0} \\ \mathbf{0}^T & 0 \end{pmatrix})$.

In addition, we notice that the Euclidean distance between the embedding $V^P(\mathbf{x})$ and $-V^L(\mathbf{v})$ is given by

$$
\begin{aligned}
&\|V^P(\mathbf{x}) - (-V^L(\mathbf{v}))\|^2 \\
&= d - 1 + \|V^P(\mathbf{x})\|^2 + 2(V^P(\mathbf{x}) \cdot V^L(\mathbf{v})) \quad (4) \\
&= d - 1 + (\|\mathbf{x}\|^2 + t^2)^2 + 2\,\mathrm{dist}^2(\mathbf{x}, l).
\end{aligned}
$$

In Equation 4, if the term $d - 1 + (\|\mathbf{x}\|^2 + t^2)^2$ is constant, then the point-to-line distance $\mathrm{dist}(\mathbf{x}, l)$ can be formalized as the distance between two points $V^P(\mathbf{x})$ and $-V^L(\mathbf{v})$ in the higher-dimensional embedded space. This is possible because t is a free variable that can be chosen arbitrarily. In particular, we choose t as a function of $\mathbf{x}$: $t(\mathbf{x}) = \sqrt{c - \|\mathbf{x}\|^2}$, where $c > \max_{\mathbf{x} \in \mathcal{D}} \|\mathbf{x}\|^2$ is a constant real value related to the entire database $\mathcal{D}$ but independent from each single item in the database. In this way, Equation 4 reduces to $\|V^P(\mathbf{x}) - (-V^L(\mathbf{v}))\|^2 = 2\,\mathrm{dist}^2(\mathbf{x}, l) + \text{constant}$.

Until now, we have successfully separated the database item (i.e., $\mathbf{x}$) from the query item (i.e., l). Next, we can pre-compute the locality-sensitive hash values for all the database items (see Section IV-D), which are used for efficient line-point k-NN computation of any given line queries. Moreover, this reduction implies that we can reduce the line-point k-NN query in a d-dimensional database $\mathcal{D}$ to a point k-NN query in a $(d+1)^2$-dimensional embedded database $V^P(\mathcal{D}) = \{V^P(\mathbf{x}_1), ..., V^P(\mathbf{x}_N)\}$, where the query item corresponds to $-V^L(\mathbf{v})$.

C. Line-point Embedding: Affine Case

Now we consider the case of any arbitrary affine line, i.e., $\mathbf{a} \neq \mathbf{0}$. Similarly to Equation 3, there is

$$
\begin{aligned}
&\mathrm{dist}^2(\mathbf{x}, l) \\
&= (\mathbf{x} - \mathbf{a}) \cdot (\mathbf{x} - \mathbf{a}) - ((\mathbf{x} - \mathbf{a}) \cdot \mathbf{v})^2 \\
&= \mathrm{Tr}((\mathbf{x} - \mathbf{a})^T(\mathbf{I} - \mathbf{v}\mathbf{v}^T)(\mathbf{x} - \mathbf{a})) \\
&= \mathrm{Tr}(\begin{pmatrix} \mathbf{x} \\ 1 \\ t \end{pmatrix}^T \underbrace{(\mathbf{I} \ -\mathbf{a} \ \mathbf{0})^T(\mathbf{I} - \mathbf{v}\mathbf{v}^T)(\mathbf{I} \ -\mathbf{a} \ \mathbf{0})}_{\mathbf{B}} \begin{pmatrix} \mathbf{x} \\ 1 \\ t \end{pmatrix}) \\
&= \mathrm{vec}(\begin{pmatrix} \mathbf{x} \\ 1 \\ t \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ 1 \\ t \end{pmatrix}^T) \cdot \mathrm{vec}(\mathbf{B}) \quad (5) \\
&= \hat{V}(\mathbf{x}) \cdot \hat{V}(\mathbf{v}, \mathbf{a}),
\end{aligned}
$$

where $\hat{V}^P(\mathbf{x})$ and $\hat{V}^L(\mathbf{v}, \mathbf{a})$ are $(d+2)^2$-dimensional embeddings for a point and line in $\mathcal{R}^d$, respectively. Similarly to Equation 4, if we choose $t(\mathbf{x}) = \sqrt{c - \mathbf{x}^2 - 1}$, where $c > \max_{\mathbf{x} \in \mathcal{D}} \|\mathbf{x}\|^2 + 1$ is a constant related to the entire database $\mathcal{D}$ (i.e., set $\|\hat{V}^P(\mathbf{x})\|^2 = c^2$), then $\mathrm{dist}^2(\mathbf{x}, l)$ also linearly

depends on the squared Euclidean distance between the embedded database and the query item: $\|\hat{V}^P(\mathbf{x}) - \hat{V}^L(\mathbf{v}, \mathbf{a})\|^2 = c^2 + d - 2 + (\mathrm{dist}^2(\mathbf{0}, l) + 1)^2 + 2\,\mathrm{dist}^2(\mathbf{x}, l)$. As a result, we can perform an affine line-point k-NN query based on a point k-NN query in a $(d+2)^2$-dimensional database $\hat{V}^P(\mathcal{D}) = \{\hat{V}^P(\mathbf{x}_1), ..., \hat{V}^P(\mathbf{x}_N)\}$, and the corresponding query item is $-\hat{V}^L(\mathbf{v}, \mathbf{a})$.

The dimension of the embedded space (i.e., $(d+1)^2$ or $(d+2)^2$) is much higher than the original space (i.e., d), and will slow down the LSH computation. We present two techniques to reduce the dimension of the embedded space.

First, notice that the matrices used within $\mathrm{vec}(\cdot)$ are symmetric matrices. For a $d \times d$ matrix $\mathbf{A}$, we can define a $d(d+1)/2$-dimensional embedding $\widehat{\mathrm{vec}}(\mathbf{A})$ as follows

$$
\widehat{\mathrm{vec}}(\mathbf{A}) = [\frac{a_{1,1}}{\sqrt{2}}, a_{1,2}, ..., a_{1,d}, \frac{a_{2,2}}{\sqrt{2}}, a_{2,3}, ..., \frac{a_{d,d}}{\sqrt{2}}]^T. \quad (6)
$$

It is easy to see that $\|\mathrm{vec}(\mathbf{A}) - \mathrm{vec}(\mathbf{B})\|^2 = 2\|\widehat{\mathrm{vec}}(\mathbf{A}) - \widehat{\mathrm{vec}}(\mathbf{B})\|^2$ and hence this dimension-reduction will not influence the accuracy of the line-point k-NN algorithm introduced above.

Secondly, we can use the Johnson-Lindenstrauss lemma [34] to reduce the dimension of the embedded data by randomly projecting the high-dimensional embedded data items onto a lower dimensional space. Compared to the first approach, this method can generate an embedding with lower dimensions, but according to our experimental results, it may reduce the accuracy of the line-point k-NN algorithms.

D. Locality-Sensitive Hash Functions for Line-Point Query

We design the hash function $\hat{h}$ for the line-point query as follows:

$$
\begin{cases}
\hat{h}(\mathbf{x}) = h(\hat{V}^P(\mathbf{x})), & \mathbf{x} \text{ is a database point} \\
\hat{h}(l) = h(-\hat{V}^L(\mathbf{v}, \mathbf{a})), & l \text{ is a line } \{\mathbf{a} + s \cdot \mathbf{v}\},
\end{cases} \quad (7)
$$

where h is a locality-sensitive hash function as defined in Section II-C. The new hash functions are locality-sensitive for line-point query, as shown by the following two theorems:

Theorem 2: The hash function family $\hat{h}$ is $(r, r(1 + \epsilon), p_1, p_2)$-sensitive if h is the hamming hash, (i.e., $h = h^{\mathbf{u}}$), where $p_1 = \frac{1}{\pi}\cos^{-1}(\frac{r^2}{C})$, $p_2 = \frac{1}{\pi}\cos^{-1}(\frac{r^2(1+\epsilon)^2}{C})$ and C is a value independent of database point, but is related to the query. Moreover, $\frac{1}{(1+\epsilon)^2} \leq \rho = \frac{\log p_1}{\log p_2} \leq 1$.

Theorem 3: The hash function family $\hat{h}$ is $(r, r(1 + \epsilon), p_1, p_2)$-sensitive if h is the p-stable hash, (i.e., $h = h^{\mathbf{a},b}$), where $p_1 = f(\frac{W}{\sqrt{2r^2+C}})$ and $p_2 = f(\frac{W}{\sqrt{2r^2(1+\epsilon)^2+C}})$ and C is a value independent of database point, but is related to the query. The function f is defined as $f(x) = \frac{1}{2}(1 - 2\,\mathrm{cdf}(-x)) + \frac{1}{\sqrt{2\pi}x}(e^{-\frac{1}{2}x^2} - 1)$, where $\mathrm{cdf}(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}t^2}dt$ is a cumulative distribution function. Moreover, $\frac{1}{1+\epsilon} \leq \rho = \frac{\log p_1}{\log p_2} \leq 1$.

The proofs of Theorem 2 and Theorem 3 are provided in Appendix A and Appendix B.

Similarly to Theorem 1 for point-point k-NN query, we can compute the error bound and time complexity for line-point k-NN query as follows:

Theorem 4: (Line-point k-NN query) Let $\mathcal{H}$ be a family of $(r, r(1+\epsilon), p_1, p_2)$-sensitive hash functions, with $p_1 > p_2$. Given a dataset of size N, we set $M = \log_{1/p_2} N$ and $L = N^\rho$, where $\rho = \frac{\log p_1}{\log p_2}$. Using $\mathcal{H}$ along with L-hash tables over M-dimensions, given a line query l, with probability at least $\frac{1}{2} - \frac{1}{e}$, our LSH algorithm solves the (r, ϵ)-neighbor problem, i.e., if there exists a point $\mathbf{x}$ that $\text{dist}(x, l) \le r(1 + \epsilon)$, then the algorithm will return the point with probability $\ge \frac{1}{2} - \frac{1}{e}$. The retrieval time is bounded by $\mathcal{O}(N^\rho)$.

The proof is given in Appendix C.

Theorem 4, along with Theorem 1, guarantees sub-linear time complexity when performing k-NN query on the historical collision results, if hamming or p-stable hashing functions are applied.

V. PROBABILISTIC COLLISION DETECTION BASED ON k-NN QUERIES

In this section, we use the LSH-based k-NN query presented in Section IV to estimate the collision probability for a given query. Our approach stores the outcome of prior instances of exact collision queries, including point queries and local path queries, within a database (shown as Approximate $\mathcal{C}_{\text{free}}$ and Approximate $\mathcal{C}_{\text{obs}}$ in Figure 1(b)). Those stored instances are used to perform probabilistic collision queries.

A. Collision Status Classifier

Our goal is to estimate the collision probability for a query point $\mathbf{p}$ or a query line l according to the database of previous collision query results. Based on the collision probability, we can design a classifier $c(\cdot)$ to predict the collision status of a given query. The expected prediction error for the classifier can be defined as

$$\mathbb{E}_{\text{error}}[c(\mathbf{p}) \mid \mathcal{D}]$$
$$= y(\mathbf{p}) \cdot \mathbb{P}[c(\mathbf{p}) = 0 \mid \mathcal{D}] + (1 - y(\mathbf{p})) \cdot \mathbb{P}[c(\mathbf{p}) = 1 \mid \mathcal{D}]$$

and

$$\mathbb{E}_{\text{error}}[c(l) \mid \mathcal{D}]$$
$$= y(l) \cdot \mathbb{P}[c(l) = 0 \mid \mathcal{D}] + (1 - y(l)) \cdot \mathbb{P}[c(l) = 1 \mid \mathcal{D}],$$

where $\mathcal{D}$, as defined before, is a dataset of N points in $\mathcal{R}^d$ and $y(\cdot)$ provides the exact collision status of $\mathbf{p}$ or l.

A classifier is *effective* at predicting the collision status of point or line queries, if its prediction error will converge to zero when the size of database $\mathcal{D}$ increases. In other words, an effective classifier $c(\cdot)$ should have the following properties:

$$\lim_{|\mathcal{D}| \to \infty} \mathbb{E}_{\text{error}}[c(\mathbf{p}) \mid \mathcal{D}] = 0 \text{ or } \lim_{|\mathcal{D}| \to \infty} \mathbb{E}_{\text{error}}[c(l) \mid \mathcal{D}] = 0.$$

As we will show in Section V-E, if a collision status classifier is effective, our probabilistic collision detection algorithm can guarantee to converge to the exact collision results, as the size of the database increases.

B. Effective Classifier for Point Query

Here we give an example implementation of an effective collision status classifier. Following the previous work on locally-weighted regression (LWR) [35], [15], we fit a Gaussian distribution to the region surrounding a query point and then estimate the probability for collision, as well as the confidence of the estimation. The confidence is further used to determine whether there is sufficient information to infer the collision status of the query, as discussed in Section V-D.

The first case is the query point, i.e., the task is to compute the collision status for a sample $\mathbf{p}$ in $\mathcal{C}$-space. We first perform point-point k-NN query to compute the prior collision instances closest to $\mathbf{p}$. Next, based on the collision status of the neighboring instances, the collision probability can be estimated as:

$$\mathbb{P}[c(\mathbf{p}) = 1 \mid \mathcal{D}] = \mathbb{E}[c(\mathbf{p}) \mid \mathcal{D}] = \mu_2 + \mathbf{\Sigma}_{12}^T \mathbf{\Sigma}_1^{-1}(\mathbf{p} - \boldsymbol{\mu}_1), \tag{8}$$

and the variance of the estimation can be given as

$$\text{Var}[c(\mathbf{p}) \mid \mathcal{D}] \tag{9}$$
$$= \frac{\Sigma_{2|1}}{(\sum_i w_i)^2} \Big(\sum_i w_i^2 + F(\mathbf{p}) \sum_i w_i^2 F(\mathbf{x}_i) \Big)$$

where $\boldsymbol{\mu}_1 = \frac{\sum_i w_i \mathbf{x}_i}{\sum_i w_i}$, $\mu_2 = \frac{\sum_i w_i y_i}{\sum_i w_i} = \frac{\sum_{\mathbf{x}_i \in S \setminus \mathcal{C}_{\text{free}}} w_i}{\sum_i w_i}$, $\mathbf{\Sigma}_1 = \frac{\sum_i w_i (\mathbf{x}_i - \boldsymbol{\mu}_1)(\mathbf{x}_i - \boldsymbol{\mu}_1)^T}{\sum_i w_i}$, $\Sigma_2 = \frac{\sum_i w_i (y_i - \mu_2)^2}{\sum_i w_i}$, $\mathbf{\Sigma}_{12} = \frac{\sum_i w_i (\mathbf{x}_i - \boldsymbol{\mu}_1)(y_i - \mu_2)}{\sum_i w_i}$, $\Sigma_{2|1} = \Sigma_2 - \mathbf{\Sigma}_{12}^T \mathbf{\Sigma}_1^{-1} \mathbf{\Sigma}_{12}$, and $F(\mathbf{x}) = (\mathbf{x} - \boldsymbol{\mu}_1)^T \mathbf{\Sigma}_1^{-1}(\mathbf{x} - \boldsymbol{\mu}_1)$. S is the neighborhood set computed using point-point k-NN query and $y_i = y(\mathbf{x}_i)$ is the exact collision status of instance $\mathbf{x}_i$. $w_i = e^{-\gamma \text{dist}(\mathbf{x}_i, \mathbf{p})}$ is the distance-tuned weight for each k-NN neighbor $\mathbf{x}_i$. The parameter γ controls the magnitude of the weight w_i, which measures the correlation between the labels of $\mathbf{x}_i$ and query point $\mathbf{p}$. In all our experiments, γ is set according to the scale of the environment (e.g., the diameter of the bounding sphere for the environment):

$$1/\sqrt{\gamma} = 0.05 \cdot \text{scale}. \tag{10}$$

Once the collision probability $\mathbb{P}[c(\mathbf{p}) = 1 \mid \mathcal{D}]$ is computed, we can predict $\mathbf{p}$'s collision status using an appropriate threshold $t \in (0, 1)$: when $\mathbb{P}[c(\mathbf{p}) = 1 \mid \mathcal{D}] > t$, we classify $\mathbf{p}$ as in-collision; otherwise, we classify it as collision-free. This classifier is effective for any $t \in (0, 1)$, because when the size of $\mathcal{D}$ increases, if $\mathbf{p}$ is actually in-collision (i.e., $y(\mathbf{p}) = 1$), more and more points in its neighborhood S will be inside $\mathcal{C}_{\text{obs}}$, and therefore $\mathbb{P}[c(\mathbf{p}) = 1 \mid \mathcal{D}]$ converges to 1. Similarly, $\mathbb{P}[c(\mathbf{p}) = 1 \mid \mathcal{D}]$ will converge to 0 if $\mathbf{p}$ is actually collision-free. As a result, given a large enough database, the classifier can always correctly predict the query point's collision status and is thus effective.

C. Effective Classifier for Local Path Query

The second case is the line query. The goal of the line query is to estimate the collision status of a local path in $\mathcal{C}$-space. We require the local path to lie within the neighborhood of the line segment l connecting its two endpoints, i.e., the local

path should not deviate too much from l. The first step is to perform a line-point k-NN query to find the prior point collision query configurations closest to the infinite line that l lies on. Next, we need to filter out the points whose projections are outside the truncated segment of l, as shown in Figure 2(c). This process might trim down some samples that are very close to the line, but lie just beyond the segment l along the axis of the line. Since these samples are isolated from the line segment by the segment's two end-points, the collision status of the segment is independent with the collision status of these samples, given that the segment's two end-points are collision-free. As a result, not considering these samples does not change the outcome of the line query. Finally, we apply our inference method (as shown below) on the filtered results, denoted as S, to estimate the collision probability of the local path.

One way to compute the collision probability for a line is to use LWR [15]. The collision probability can be estimated as:

$$\mathbb{P}[c(l) = 1 \mid \mathcal{D}] = \mathbb{E}(c(l) \mid \mathcal{D}) \tag{11}$$
$$= \mu_2 + \boldsymbol{\Sigma}_{12}^T \boldsymbol{\Sigma}_1^{-1} (\texttt{NearestPnt}(l, \boldsymbol{\mu}_1) - \boldsymbol{\mu}_1),$$

and

$$\text{Var}[c(l) \mid \mathcal{D}] \tag{12}$$
$$= \frac{\Sigma_{2|1}}{(\sum_i w_i)^2} \left(\sum_i w_i^2 + F(\texttt{NearestPnt}(l, \boldsymbol{\mu}_1)) \sum_i w_i^2 F(\mathbf{x}_i) \right).$$

where the symbols are as defined in Equation 8 and Equation 12, except the terms related with w_i, which is now defined as $w_i = e^{-\gamma \, \text{dist}(\mathbf{x}_i, l)}$. Function $\texttt{NearestPnt}(l, \mathbf{x})$ returns a point on line segment l that is closest to a point $\mathbf{x}$.

However, the above LWR-based method has some limitations. The main issue is that it can only compute a collision probability for the entire line. In many cases, we need to know where the collision is likely to happen on the line (i.e., the first time of contact (TOC)). We provide an optimization method for estimating the approximate TOC. In particular, we divide the line l into I segments and assign each segment, say l_i, a label c_i to indicate its collision status. We aim to find a suitable label assignment $\{c_i^*\}_{i=1}^I$ so that:

$$\{c_i^*\} = \text{argmin}_{\{c_i\} \in \{0,1\}^I} \sum_{i=1}^I (c_i - c_i')^2 + \kappa \sum_{i=1}^{I-1} (c_i - c_{i+1})^2,$$

where c_i' is the collision status for the midpoint of l_i estimated using Equation 8. The term $(c_i - c_i')^2$ constrains the label assignment to be consistent with point query results, and $\sum_{i=1}^{I-1} (c_i - c_{i+1})^2$ is a smoothness term, which models the fact that collision labels for adjacent points are likely to be the same. Parameter κ adjusts the relative weight between the consistency term and the smoothness term. The optimization can be computed efficiently using dynamic programming. After that, we can estimate the collision probability for the line as

$$\mathbb{P}[c(l) = 1 \mid \mathcal{D}] = \mathbb{E}(c(l) \mid \mathcal{D}) = \max_{i: \, c_i^*=1} c_i', \tag{13}$$

and the approximate first time of contact can be given as $\min_{i: \, c_i^*=1} i/I$.

Based on the collision probability formulated as above, we can design a classifier to predict the collision status for a given line query by using a specific threshold $t \in (0,1)$ to justify whether the query is in-collision or not. If the query's collision probability is larger than t, we return in-collision; otherwise, we return collision-free. This classifier is also effective for any $t \in (0,1)$, because when the size of $\mathcal{D}$ increases, if l is in-collision, there always exists one segment l_i on l whose collision probability c_i' converges to 1 and therefore $\mathbb{P}[c(l) = 1 \mid \mathcal{D}]$ will converge to 1. Similarly, if l is collision-free, the probability will converge to 0.

Remark The collision status classifiers described above are generative classifiers, i.e., they are constructed after the conditional collision probability is computed. One advantage of the generative classifier is that it can be used even in a dynamic environment where the obstacles may change their positions. However, in our approach, we only need to know the binary collision status of the query instead of its collision probability. As a result, we can use effective *discriminative* classifiers, i.e., design a classifier directly from the data. For example, we can use the weighted average of the query's neighbors' collision status to predict the query's collision status; then all we need to learn are those weight factors. Given a large database of historical data, a discriminative classifier is usually more robust than a generative classifier. However, the discriminative classifier is specific to the current database, and the need to learn a new classifier when the environment changes can be expensive. The discriminative classifier is thus limited to static environments.

D. Rejection Rules

When using the methods discussed above to estimate the collision status for a given point or line query, there must be sufficient number of data items surrounding the query to give an estimate with a high level of confidence. Otherwise, we should reject the estimated collision status and rather perform exact collision checking on the query.

We consider two types of rejection rules [33]: *ambiguity rejection* and *distance rejection*.

- Ambiguity rejection happens when the estimated collision status is ambiguous. For instance, suppose there are the same number of in-collision points and collision-free points in the neighborhood of a point query $\mathbf{p}$, and these points all lie same distance from the query. The collision probability computed by Equation 8 is 0.5 in this case; therefore any estimate of the collision status is equivalent to a random guess. Ambiguity also occurs when the variance of the estimated collision status (computed by Equation 9) is large. To determine whether ambiguity rejection is necessary for a point query $\mathbf{p}$, we measure the ambiguity as

$$\texttt{Amb} = \big(\min(\mathbb{E}[c(\mathbf{p})], 1 - \mathbb{E}[c(\mathbf{p})]) \big)^2 + \text{Var}[c(\mathbf{p})],$$

where $\mathbb{E}[c(\mathbf{p})]$ and $\text{Var}[c(\mathbf{p})]$ are computed according to Equation 8 and Equation 9. If $\texttt{Amb}$ is larger than a given
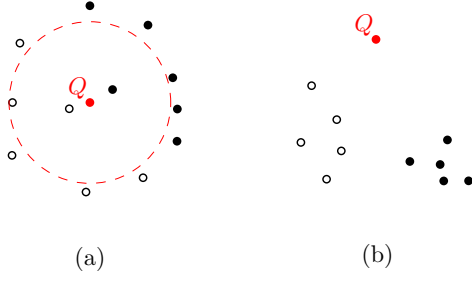
Fig. 3: Two rejection rules: (a) ambiguity rejection: Q's estimated collision probability is near 0.5 or the variance for the estimate is large; (b) distance rejection: when Q is far from all in-collision and collision-free database items.

threshold A_d, we reject the estimate and perform the exact collision test.

- Distance rejection happens when the k-NN points for a given query lie too far away from the query configuration (in terms of the distance). This is a problem because our collision status estimator is based on coherency of nearby points' or lines' collision statuses. This distance rejection happens when the database is nearly empty, or when the query is in a region not well sampled by current configuration database. In order to determine whether we need to perform distance rejection, we compute `Dis`, the distance from $\mathbf{p}$ to its nearest point. If `Dis` is larger than a given threshold D_d (for instance, `Dis` is ∞ when the database is empty), we perform the exact collision query.

The two rejection rules are shown in Figure 3. The rejection rules for a line query are similar.

E. Asymptotic Property of Probabilistic Collision Query

If the classifier used in the probabilistic collision query is effective, we can prove that the collision status returned by the probabilistic collision checking module will converge to the exact collision detection results when the size of the dataset increases (asymptotically):

Theorem 5: The collision query performed using LSH-based k-NN will converge to the exact collision detection as the size of the dataset increases.

Proof: We only need to prove that both the probability of a false positive (i.e., returns in-collision status when there is in fact no collision) and a false negative (i.e., returns collision-free when there is in fact a collision) converges to zero, as the size of the database increases.

Given a query, we denote its r-neighbor as B_r, where r is the distance between the query and its k-th nearest neighbor. For a point query, B_r is an r-ball around it. For a line query, B_r is the set of all points with distance r to the line (i.e., a line swept-sphere volume). Let $P_1 = \frac{\mu(B_{r(1+\epsilon)} \cap \mathcal{C}_{\text{obs}})}{\mu(\mathcal{C}\text{-space})}$ and $P_2 = \frac{\mu(B_{r(1+\epsilon)} \cap \mathcal{C}_{\text{free}})}{\mu(\mathcal{C}\text{-space})}$, which are the probabilities that a uniform sample in $\mathcal{C}$-space is in-collision or collision-free and within query's $r(1+\epsilon)$-neighborhood. Here $\mu(\cdot)$ is the volume measure. Let N be the size of the database corresponding to the prior instances.

A false negative occurs if and only if the following two cases are true: 1) there are no in-collision points within $B_{r(1+\epsilon)}$,

and therefore the probabilistic method always returns collision-free; 2) there are in-collision points within $B_{r(1+\epsilon)}$, but the classifier predicts wrong label.

First, we compute the probability for case 1. The event that there are no in-collision points within $B_{r(1+\epsilon)}$ happens either when no dataset point lies within $B_{r(1+\epsilon)}$ or when there exist some points within that ball which are missed due to the approximate nature of LSH-based k-NN query. According to Theorem 1, we have

$$\mathbb{P}[\text{case 1}]$$
$$= \sum_{i=0}^{N} \binom{N}{i}(1-P_1)^{N-i}P_1^i(1-(1/2-1/e))^i$$
$$= (1 - P_1(1/2 - 1/e))^N \to 0 \text{ (as } N \to \infty).$$

Case 2 can occur when case 1 does not happen and the classifier gives the wrong results. However, as the classifier is effective, we have

$$\mathbb{P}[\text{case 2}]$$
$$= (1 - \mathbb{P}[\text{case 1}]) \cdot \mathbb{P}_{\text{error}}[\mathbf{x} \text{ or } l \text{ in-collision}; \mathcal{D}]$$
$$= (1 - \mathbb{P}[\text{case 1}]) \cdot \mathbb{E}_{\text{error}}[c(\mathbf{x}) \text{ or } c(l) \mid \mathcal{D}]$$
$$\to 0 \text{ (as } N \to \infty).$$

As a result, we have

$$\mathbb{P}[\text{false negative}] = \mathbb{P}[\text{case 1}] + \mathbb{P}[\text{case 2}] \to 0 \text{ (as } N \to \infty).$$

Similarly, a false positive occurs if there are no collision-free points within $B_{r(1+\epsilon)}$ or if there are collision-free points within $B_{r(1+\epsilon)}$ but the classifier still predicts a wrong label. The probability of case 1 can be given as

$$\mathbb{P}[\text{case 1}] = (1 - P_2(1/2 - 1/e))^N$$

and the probability of case 2 is

$$\mathbb{P}[\text{case 2}] = (1 - \mathbb{P}[\text{case 1}]) \cdot \mathbb{P}_{\text{error}}[\mathbf{x} \text{ or } l \text{ collision free}; \mathcal{D}]$$
$$= (1 - \mathbb{P}[\text{case 1}]) \cdot \mathbb{E}_{\text{error}}[c(\mathbf{x}) \text{ or } c(l) \mid \mathcal{D}].$$

Both terms converge to zero when the size of the database increases. As a result, we can conclude that the false positive also converges to 0:

$$\mathbb{P}[\text{false positive}] = \mathbb{P}[\text{case 1}] + \mathbb{P}[\text{case 2}] \to 0 \text{ (as } N \to \infty).$$
$$\blacksquare$$

Remark Note that the convergence of the collision query using LSH-based k-NN query is slower than that using the exact k-NN based method, whose prediction errors can be given as: $\mathbb{P}[\text{false negative}] = (1 - P_1)^N$ and $\mathbb{P}[\text{false positive}] \leq (1 - P_2)^N$.

Remark The fact that the probability of getting false negative (or false positive) converges to 0 is true if and only if P_1 (or P_2) is not 0. Usually we assume that real world obstacles are compact and therefore obstacles in $\mathcal{C}$-space are also compact. Thus, if a configuration is collision-free, there is an open set surrounding it that is collision-free as well and therefore $P_2 > 0$. However, a configuration in-collision (i.e., inside a contact set) does not necessarily have a positive P_1 (i.e., P_1 may be zero). As these kinds of 'bad' configurations are of zero measure, our proof of Theorem 5 still holds.

VI. Accelerating Sampling-based Planners

In this section, we first discuss techniques to accelerate various sampling-based planners using our probabilistic collision query, including 1) how the database is constructed and maintained; 2) how to accelerate various planners; 3) how to handle dynamic environments; 4) how to combine these techniques with non-uniform sampling techniques. Next, we analyze the factors that can influence the performance of resulting planners using our probabilistic collision queries. Finally, we prove the completeness and optimality of modified sampling-based planners.

A. Database Construction

When the planner thread starts, the database of prior collision query results is empty. Given a point query, we first compute its k-nearest neighboring points S. Based on S, we check whether distance rejection is necessary. If so, we perform exact collision test and add the query result into the database. Otherwise, we estimate the query's collision probability and the confidence of our estimate, using approaches discussed in Section V. Next, we check for ambiguity rejection. Based on the outcome of ambiguity rejection, we may again perform exact collision query and add the result to the database; or the estimated collision result can be directly used by a sampling-based planner. When a local path query is given, the processing pipeline is similar, except that when performing exact collision checking of the local path, a series of point configurations on the local path are added to the database. In summary, we perform exact collision tests only for queries that are located within regions that are not well covered by the current database D; the resulting query results are added into $\mathcal{D}$. Later, in Section VI-B, we verify the collision status of a query using exact collision test when it is estimated as collision-free. This test is performed to guarantee the overall motion planning algorithm to be conservative. However, such queries are not added to the database.

Next, we discuss the efficiency of operations on the LSH-based database, which is implemented as a hash table. The hash table starts out empty, so there is no pre-processing overhead. When we decide to add the result for a collision query $\mathbf{x}$ into the database, we first compute its hashing code $\hat{h}(\mathbf{x})$ and then add it into the hash table. This step's complexity remains constant. After warm-up, we begin performing k-NN query on the hash table, which has the complexity $\mathcal{O}(N^\rho)$ (all symbols are as defined in Theorem 4). After adding N items into the hash table and performing M probabilistic collision queries, the overall complexity of the database operations becomes $\mathcal{O}(N + M \cdot N^\rho)$. Note that the number of all collision queries is larger than $\max(N, M)$; therefore the amortized computational overhead on each collision query is $\mathcal{O}(1)$.

B. Accelerating Various Planners

Algorithm 1 highlights our basic approach to apply the probabilistic collision query: we use the computed collision probability as a filter to reduce the number of exact collision queries. If a given configuration or local path query is close

(a) I-PRM

```
sample(𝒟ᵒᵘᵗ, n)
V ← 𝒟 ∩ 𝒞_free, E ← ∅
foreach v ∈ V do
    U ← near(G^{V,E}, v, 𝒟ⁱⁿ)
    foreach u ∈ U do
        if icollide(v, u, 𝒟ⁱⁿ) < t
            if ¬collide(v, u, 𝒟ᵒᵘᵗ)
                E ← E ∪ (v, u)
```

near: nearest neighbor search.
icollide: probabilistic collision checking based on k-NN.
collide: exact local path collision checking.
$\mathcal{D}^{\text{in/out}}$: prior instances as input/output.

(b) I-lazyPRM

```
sample(𝒟ᵒᵘᵗ, n)
V ← 𝒟 ∩ 𝒞_free, E ← ∅
foreach v ∈ V do
    U ← near(G^{V,E}, v, 𝒟ⁱⁿ)
    foreach u ∈ U do
        w ← icollide(v, u, 𝒟ⁱⁿ)
        l ← ‖(v, u)‖
        E ← E ∪ (v, u)^{w,l}
    do
        search path p on G(V, E) which minimizes
            ∑_e l(e) + λ min_e w(e).
        foreach e ∈ p, collide(e, 𝒟ᵒᵘᵗ)
    while p not valid
```

(c) I-RRT

```
V, 𝒟 ← x_init, E ← ∅
while x_goal not reach
    x_rnd ← sample-free(𝒟ᵒᵘᵗ, 1)
    x_nst ← inearst(G^{V,E}, x_rnd, 𝒟ⁱⁿ)
    x_new ← isteer(x_nst, x_rnd, 𝒟^{in,out})
    if icollide(x_nst, x_new) < t
        if ¬collide(x_nst, x_new)
            V ← V ∪ x_new, E ← E ∪ (x_new, x_nst)
```

inearest: find the nearest tree node that has high collision-free probability.
isteer: steer from a tree node to a new node, using icollide for validity checking.

(d) I-RRT*

```
V, 𝒟 ← x_init, E ← ∅
while x_goal not reach
    x_rnd ← sample-free(𝒟ᵒᵘᵗ, 1)
    x_nst ← inearst(G^{V,E}, x_rnd, 𝒟ⁱⁿ)
    x_new ← isteer(x_nst, x_rnd, 𝒟^{in,out})
    if icollide(x_nst, x_new) < t
        if ¬collide(x_nst, x_new)
            V ← V ∪ x_new
            U ← near(G^{V,E}, x_new)
            foreach x ∈ U, compute weight c(x) =
                λ‖(x, x_new)‖ + icollide(x, x_new, 𝒟ⁱⁿ)
            sort U according to weight c.
            Let x_min be the first x ∈ U with ¬collide(x, x_new)
            E ← E ∪ (x_min, x_new)
            foreach x ∈ U, rewire(x)
```

inearest: find the nearest tree node that has high collision-free probability.
isteer: steer from a tree node to a new node, using icollide for validity checking.
rewire: RRT* routine used to update the tree topology for optimality guarantee.

Fig. 4: Our probabilistic collision checking module can improve a wide variety of motion planners. Here we present four modified planners as example.

to in-collision instances, then it has a high probability of being in-collision. Similarly, if a query has many collision-free instances around it, it is likely to be collision-free. In our implementation, we cull away only those queries with high collision probabilities. For queries with high collision-free probability, we still perform exact collision tests on them in order to guarantee that the overall collision detection algorithm is conservative. In Figure 4(a), we show how our probabilistic culling strategy can be integrated with the PRM algorithm by only performing exact collision checking (`collide`) for queries with collision probability (`icollide`) larger than a given threshold t. Note that the neighborhood search routine (`near`) can use LSH-based point-point k-NN query. `icollide` is computed according to Equation 8 or Equation 11.

In Figure 4(b), we show how to use the collision probability as a cost function with the lazyPRM algorithm [1]. In the basic version of lazyPRM algorithm, the expensive local path collision checking is delayed till the search phase. The basic idea is that the algorithm repeatedly searches the roadmap to compute the shortest path between the initial and goal nodes, performs collision checking along the edges, and removes the in-collision edges from the roadmap. However, the shortest path usually does not correspond to a collision-free path, especially in complex environments. We improve the lazyPRM planner using probabilistic collision queries. We compute the collision probability for each roadmap edge during roadmap construction, based on Equation 13. The probability (w) as well as the length of the edge (l) are stored as costs of the edge. During the search step, we try to compute the shortest path with a minimum collision probability, i.e., a path that minimizes the cost $\sum_e l(e) + \lambda \min_e w(e)$, where λ is a parameter that controls the relative weight of path length and collision probability. As the prior knowledge about obstacles is implicitly taken into account based on collision probability, the resulting path is more likely to be collision-free.

Finally, the collision probability can be used by the motion planner to explore $\mathcal{C}_{\text{free}}$ in an efficient manner. We use RRT to illustrate this benefit (Figure 4(c)). Given a random sample x_{rnd}, RRT computes a node x_{nst} among the prior collision-free configurations that are closest to x_{rnd} and expands from x_{nst} towards x_{rnd}. If there is no obstacle in $\mathcal{C}$-space, this exploration technique is based on the Voronoi heuristic that biases the planner towards the unexplored regions. However, the existence of obstacles affects its performance: the planner may run into $\mathcal{C}_{\text{obs}}$ shortly after expansion, and the resulting exploration is limited. Using the k-NN based inference, we can estimate the collision probability for local paths connecting x_{rnd} with each of its neighbors and choose x_{nst} as the one with both a long edge length and a small collision probability (i.e., $x_{\text{nst}} = \arg\max(l(e) - \lambda \cdot w(e))$, where λ is a parameter used to control the relative weight of these two terms). A similar strategy can also be used for RRT*, as shown in Figure 4(d).

C. Narrow Passages and Non-uniform Samples

Narrow passages are a key issue for sampling-based motion planners. In general, it is difficult to generate enough number of samples in the narrow passages and capture the connectivity of the free space. Narrow passages can lead to some additional issues in terms of the collision status classifier presented in SectionV-B. In narrow passages, a collision-free query point configuration can be wrongly classified as in-collision, as shown in Figure 5. This is because the collision status inference algorithm assumes the spatial coherency about the collision status, i.e., nearby samples in the $\mathcal{C}$-space tend to have the same collision status. However, such spatial coherency may not work in the regions around narrow passages and this may reduce the accuracy of collision status estimated via the inference algorithm. This reduced accuracy can also decrease the performance of the sampling-based planner in narrow passages. If a random planner can indeed generate a free-space sample in the narrow passage, the inference algorithm may incorrectly classify it as in-collision and may not add it to the database $\mathcal{D}$ and the roadmap/tree-structure computed by the planner. As a result, the planner may not be able to capture the connectivity of the space correctly around that narrow passage.
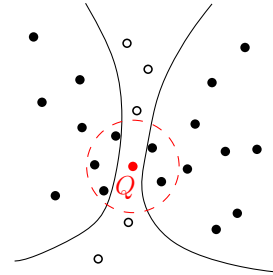

Fig. 5: Q is a collision-free query point configuration inside the narrow passage, but the collision status classifier may estimate it as in-collision, because all its nearby samples in the database (shown as black dots) are all in-collision.

One solution to address this problem is to perform a double check on a query's collision status occasionally using exact collision test, even when the inference algorithm estimates the query to have a large collision probability. In particular, suppose the estimated collision probability of a query is p, where $p \in (0.5, 1]$, then with a probability of $\max(1 - p, p_s)$, we check the exact collision status of the query, where p_s is a small value (e.g., 0.01). In practice, this occasional verification strategy works well on narrow passage benchmarks and provides a good trade-off between efficiency and completeness.

However, for some narrow passages with small expansiveness, defined based on the criteria in [36], this occasional verification with an exact collision test can slowdown the process of generating sufficient number of samples in the narrow passages, which affects the performance of probabilistic collision checking. In order to handle challenging narrow passage scenarios, we combine the non-uniform sampling strategies used in different sampling-based planners [3], [5], [6] with our probabilistic collision query to quickly generate more narrow passage samples in the database $\mathcal{D}$ and the planner's roadmap. In particular, with a probability of $1 - p_s$, we perform uniform sampling using probabilistic collision checking; with a probability of p_s, we perform non-uniform sampling with exact collision checking to increase the number of samples in the narrow passages. The samples generated by

non-uniform sampling are directly added into the database and are used later by the inferencing algorithm.

D. Performance Analysis

The modified planners are faster, mainly because we replace some of the expensive, exact collision queries with relatively cheap k-NN queries. Let the timing cost for a single exact collision query be T_C and for a single k-NN query be T_K, where $T_K < T_C$. Suppose the original planner performs C_1 collision queries and modified planners performs C_2 collision queries and $C_1 - C_2$ k-NN queries, where $C_2 < C_1$. We also assume that the two planners spend the same time A on other computations within a planner, such as sample generation, maintaining the roadmap or the tree structure, etc. Then the speedup ratio obtained by the modified planner is:

$$R = \frac{T_C \cdot C_1 + A}{T_C \cdot C_2 + T_K \cdot (C_2 - C_1) + A}. \quad (14)$$

Therefore, if $T_C \gg T_K$ and $T_C \cdot C_1 \gg A$, we have $R \approx C_1/C_2$, i.e., if the higher number of exact collision queries are culled, we can obtain a higher speedup. The extreme speedup ratio C_1/C_2 may not be reached, however, for two reasons. 1) $T_C \cdot C_1 \gg A$ may not hold, such as when the underlying collision-free path solution lies in some narrow passages (A is large) or in open spaces ($T_C \cdot C_1$ is small); or 2) $T_C \gg T_K$ may not hold, such as when the environment and robot have low geometric complexity (i.e., T_C is small) or the instance dataset is large and the cost of the resulting k-NN query is high (i.e., T_K is large).

Note that R is only an approximation of the actual acceleration ratio. It may overestimate the speedup, because a collision-free local path may have a collision probability higher than a given threshold; our probabilistic collision approach filters such high probabilities out. If such a collision-free local path is critical for the connectivity of the roadmap, such false positives due to the probabilistic collision checking module will cause the resulting planner to perform more exploration, and thereby increases the overall planning time. As a result, we need to choose an appropriate threshold that can provide a balance: we need a large threshold to filter out more collision queries and increase R; at the same time, we need to use a small threshold to reduce the number of false positives. However, the threshold choice is not important in the asymptotic sense. According to Theorem 5, the false positive error converges to 0 when the database size increases.

R may also underestimate the actual speedup, because the timing cost for different collision queries can be different. For configurations near the boundary of $\mathcal{C}_{\text{obs}}$, the collision queries are more expensive. Therefore, the timing cost of checking the collision status for an in-collision local path is usually larger than that of checking a collision-free local path, because the former always has one configuration on the boundary of $\mathcal{C}_{\text{obs}}$. As a result, it is possible to obtain a speedup larger than C_1/C_2.

E. Completeness and Optimality

As a natural consequence of Theorem 5, we can prove the probabilistic completeness and optimality of the new planners.

To avoid the narrow passage problems, while discussing the new planners' completeness, we assume that they apply the heuristics mentioned in Section VI-C. In other words, we assume $p_s > 0$ in order to guarantee that the critical samples in the narrow passage will not be filtered out by mistake.

Theorem 6: I-PRM and I-lazyPRM are probabilistically complete. I-RRT* is probabilistically complete and asymptotically optimal.

Proof: A motion planner MP is probabilistically complete if its failure probability, i.e., when a collision-free path exists, the probability that it cannot find a solution after N samples converges to 0 when the number of samples N increases: $\lim_{N \to \infty} \mathbb{P}[\text{MP fails}] = 0$, where [MP fails] denotes the event that the motion planner fails to find a solution after N samples, when the solution exists.

Suppose we replace MP's exact collision detection query by the probabilistic collision query and denote the new planner as I-MP. I-MP can fail in two cases: 1) MP fails; 2) MP computes a solution but some edges on the collision-free path are classified as in-collision by our collision status inference algorithm (i.e., false positives). Let L be the number of edges in the solution path and let E_i denote the event that the i-th edge is incorrectly classified as in-collision. As a result, we have

$$\mathbb{P}[\text{I-MP fails}]$$
$$= \mathbb{P}[\text{MP fails}] + (1 - \mathbb{P}[\text{MP fails}]) \cdot \mathbb{P}[\bigcup_{i=1}^{L} E_i]$$
$$\leq \mathbb{P}[\text{MP fails}] + \sum_{i=1}^{L} \mathbb{P}[E_i].$$

Similar to [MP fails], the event [I-MP fails] denotes the event that the new motion planner fails to find a solution after N samples, when the solution exists. According to Theorem 5, $\lim_{N \to \infty} \mathbb{P}[E_i] = 0$ and L is a finite number, then we have $\lim_{N \to \infty} \mathbb{P}[\text{I-MP fails}] = 0$, i.e., I-MP is probabilistically complete. Therefore, as PRM, lazyPRM and RRT* are all probabilistically complete, we can prove that I-PRM, I-lazyPRM and I-RRT* are all probabilistically complete.

Similarly, if MP is asymptotically optimal, then I-MP may not converge to the optimal path only when some of the path edges are classified as in-collision by the collision status inference algorithm and this probability converges to zero. As a result, I-RRT* is asymptotically optimal. ∎

We did not include I-RRT in the above theorem, because the theorem and its proof may not directly apply to I-RRT. In particular, the proof implicitly requires the motion planner to converge to a solution that is independent of the order of the samples. Such requirement is satisfied by roadmap-based approaches such as PRM and lazyPRM, and is also satisfied by asymptotically optimal algorithms such as RRT*. However, RRT has the characteristic that the order of the samples can make a difference in terms of probabilistic completeness guarantees. Since our probabilistic collision checking can change the generating order of the samples in the free configuration space, the collision-free samples in the I-RRT tree may be added in some particular sequences that cannot guarantee

probabilistic completeness. In particular, in Figure 4(c), the I-RRT tree may choose to extend from a node x_{nst} that is not closest to the random x_{rnd}, because the actual nearest node may have a higher collision probability. This variant in adding new samples in the I-RRT algorithm may break one of the properties of the original RRT algorithm: for any sample in the free configuration space, its distance to the RRT tree will converge in probability to zero (Lemma 2 in [2]). This property implies that the RRT tree would cover the entire free space in the limit, and is important for proving the probabilistic completeness of RRT. In order to prove the probabilistic completeness of I-RRT, we either need to show that this property still holds for I-RRT, or we need to find other methods to directly prove that the I-RRT tree will eventually cover the entire free configuration space.

Even though we are unable to guarantee the probabilistic completeness of I-RRT, our current implementation of I-RRT works well in our benchmarks as shown in Table I and Table II. This is probably due to the fact that the randomness of our probabilistic checking algorithm ensures that such pathological cases of sequences do not arise in our current benchmarks. As a result, in most cases I-RRT should work well.

F. Speed and Completeness

The performance improvement in the new sampling-based planners is achieved by using cheaper hash table-based probabilistic collision queries to replace the expensive exact collision checking queries. In order to guarantee probabilistic completeness, we need to optionally check the false positive errors of probabilistic collision checking and/or use non-uniform sampling to overcome the difficulty in narrow passages, as discussed in Section VI-C. This implies that to guarantee probabilistic completeness, we need to perform extra tests which actually degrade the runtime performance of our motion planner. As a result, we have a tradeoff between runtime efficiency and guarantees of probabilistic completeness. In particular, when a planner is slow in terms of making progress towards the goal configuration, which is probably due to narrow passages, we set a higher priority for completeness and increase the probability to use the narrow passage sampling strategy discussed in Section VI-C; when a planner is moving quickly towards the goal position, which implies open spaces with good visibility, we set a higher priority for efficiency and decrease the probability of using the narrow passage sampling strategy.

VII. RESULTS AND DISCUSSIONS

In this section, we highlight the performance of sampling-based planners with the probabilistic collision checking module. Figure 6 and Figure 7 show the articulated PR2 and rigid body benchmarks we used to evaluate the performance. We evaluate each planner on different benchmarks. For each combination of planner and benchmark we ran 50 instances of the planner, and computed the average planning time as an estimate of the planner's performance on this benchmark. The algorithm is implemented in C++ and all the experiments are performed on a PC with an Intel Core i7 3.2GHz CPU with

2GB memory. The exact collision tests are performed using FCL collision library [37].

A. Pipeline and Results

While using the probabilistic collision checking module, the planners have a 'cold start', i.e., they start with an empty database of prior collision query results, which means that they have no knowledge about the environment in the beginning. As a result, during the first few queries performed by sampling-based planner, the probabilistic collision framework will find that it does not have sufficient information to predict the collision status of a given configuration or a local path. In these cases, we end up using exact collision checking algorithms. During this phase, the modified planner will behave exactly the same as the original planner, except that the results from exact collision queries will be stored in the database. This process is called the 'warm up' of the modified planning framework. After several planning queries, there will be enough information in the database about $\mathcal{C}$-space to perform inference, and the acceleration brought by the probabilistic collision checking method begins to counteract its overhead during the following queries.

The comparison results are shown in Table I and Table II, corresponding to PR2 benchmarks and rigid body benchmarks, respectively. Based on these benchmarks, we observe that:

- The usage of probabilistic collision checking module results in more planning speedup on articulated models than on rigid body benchmarks. Exact collision checking on articulated models is more expensive than exact collision checking on rigid models, because for articulated models we need to compute self-collision as well as check for collisions between each component of the body and each obstacle in the environment. This makes T_C larger and results in larger speedups.
- The speedup of I-PRM over PRM is relatively large, since exact collision checking takes a significant fraction of overall time within PRM algorithm. I-lazyPRM also provides good speedup as the candidate path is nearly collision-free and can greatly reduce the number of exact collision queries in lazy planners. The speedups of I-RRT and I-RRT* are limited or can even be slower than the original planners, especially on simple rigid body benchmarks.
- On benchmarks with narrow passages, our approach does not increase the probability of finding a solution. However, probabilistic collision checking is useful in culling some of the colliding local paths.
- The variance of planning time does not change much between modified planners and the corresponding original planners.

On most benchmarks in Table I and Table II, the performance improvement of leveraging the probabilistic collision checking is higher on multi-query planners (PRM and lazyPRM) than multi-query planners (RRT and RRT*). This difference in performance is due to the different sampling criteria used in single query v.s. multi-query planners. While multi-query planners like PRM and lazyPRM tend to construct

a roadmap with many samples to cover the free space, single-query planners only generate samples that are just enough to find a solution trajectory connecting the initial and goal. As a result, in the beginning stage of a single-query planner, the probabilistic collision checking may result in a high number of false positives due to the lack of sufficient number of samples in the database, which will degrade the performance of single-query planners.

We also observe that on several benchmarks without narrow passages (i.e., "apartment" and "easy" in Table II), I-RRT and I-RRT* can even result in worse performance, as compared to RRT and RRT*, respectively. Such reduced performance occurs due to the following reasons. First, the original RRT and RRT* planners are already quite efficient on these easy benchmarks, while the database operations in the k-NN inference framework result in computational overhead. Second, both RRT and RRT* are single-query planners and thus the probabilistic collision checking may result in a high number of false positives.

On benchmarks with narrow passages (i.e., "flange" and "torus" in Table II), I-RRT and I-RRT* can always find a solution, and are more efficient than the original planners RRT and RRT*, respectively. This is because single-query planners need to generate a sufficient number of samples in the free space to capture the connectivity of the region around the narrow passages. The large number of samples is important for the performance of I-RRT and I-RRT*. First, it decreases the probabilistic collision checking's false negative rate around the narrow passages, and this is crucial for I-RRT and I-RRT* to find a solution in narrow passages. Second, it results in a large speedup ratio as defined in Equation 14, because the time cost of large number of exact collision checking makes the overhead of database operations negligible.

We also analyze the accuracy of our collision status estimation algorithm in Figure 8. For databases of different sizes, we compute the average inference accuracy on 100 point queries or local path queries. If an ambiguity rejection or a distance rejection happens for a query, we measure the inference accuracy for this query as 0.5 because any estimate is no better than random guess, and we in fact perform exact collision checking on such samples. This explains why the inference accuracy is 0.5 when the database is empty. From the result, we can see the inference accuracy increases when the database becomes larger.

As the planners with our probabilistic collision checking use a 'cold start', we need to point out that the acceleration results in Table I and Table II demonstrate only part of the speedups that can be obtained using our approach. As more collision queries are performed and their results are stored in the dataset, the resulting planner has more information about $\mathcal{C}_{\text{obs}}$ and $\mathcal{C}_{\text{free}}$, and becomes more efficient in terms of culling. Ideally, we can filter out all in-collision queries and obtain a high speedup. In practice, we do not achieve ideal speedups due to two reasons: 1) we only have a limited number of samples in the dataset; 2) the overhead of the k-NN query increases as the dataset size increases. As a result, when we perform the global motion planning computation repeatedly, the planning time will first decrease to a minimum, and then

increase. This phenomenon is shown in Figure 9.

To alleviate the downgrading of the planner's performance while the database size increases, our solution is only adding query results that are potential to be useful. For instance, if a query configuration is near the boundary of $\mathcal{C}_{\text{obs}}$, then there would be similar number of in-collision and collision-free samples in its k-nearest neighbors (Figure 3(a)), and hence the ambiguity rejection tends to happen and the query will be added into the database $\mathcal{D}$ after exact collision test. This implies that $\mathcal{D}$ has a bias for points near boundaries. However, such points may not be useful if the sampling density around an obstacle is already high. As a result, we can choose to avoid adding such configurations to the database, when its distance to the farthest k-NN neighbor is smaller than a given threshold. The result of this strategy is shown in Figure 10. It is obvious that average planning time first decreases and is unchanged after that. Our current way of choosing which configurations to incorporate into the database is a heuristic. In the future, it would be worthwhile to use more sophisticated approaches such as [38], which used the concept of information gain to evaluate a configuration's potential usefulness for the motion planning.

Our approach is also memory efficient. In practice, the memory consumed by our modified motion planners is about 2-3 times more (due to the database) than the original planners.

Sensitivity Analysis for Parameters: We also perform a simple sensitivity analysis for parameters γ and t of the collision status classifiers. As we discussed in Section V-A, γ is used to adjust the contribution of each k-NN sample in the final collision status estimation, and t measures the planner's confidence about the results computed using estimation. Different settings of these two parameters will change the probabilistic collision checking result and eventually influence the planning behavior. In our sensitivity test, we deviate these parameters $\pm 20\%$ from the default values used in our experiments, where the default value for γ is as defined in Equation 10 and the default value of t is 0.2. We find such deviation does not change the success/failure of the planning algorithms, and the change to the planner's timing performance is within $\pm 10\%$.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we use probabilistic collision checking to improve the performance of sampling-based motion planners. The basic idea is to store the prior collision results as an approximate representation of $\mathcal{C}_{\text{obs}}$ and $\mathcal{C}_{\text{free}}$, and to replace the expensive exact collision detection query by a relatively cheap probabilistic collision query. We integrate probabilistic collision routines with various sampling-based motion planners and observe 30% to 2x speedup on rigid and articulated robots.

There are many avenues for future work. First, we need to find methods to adjust LSH parameters adaptively so that the k-NN query becomes more efficient for varying dataset sizes. One possible way is to change L (the number of hash tables), because a small L may provide sufficient k-NN candidates for a large dataset. Secondly, for samples in regions

	PRM	I-PRM	lazyPRM	I-lazyPRM	RRT	I-RRT	RRT*	I-RRT*
(a)	12.78/0.49	9.61/1.15 (32%)	1.2/0.54	0.87/0.43 (37%)	0.96/0.34	0.75/0.35 (28%)	1.12/0.5	1.01/0.52 (11%)
(b)	23.7/6.25	12.1/4.3 (96%)	1.7/1.08	0.90/0.64 (88%)	1.36/0.85	0.89/0.46 (52%)	2.08/1.46	1.55/0.75 (34%)
(c)	fail	fail	fail	fail	4.15/2.23	2.77/1.18 (40%)	3.45/2.12	2.87/1.53 (20%)
(d)	18.5/8.3	13.6/6.62 (36%)	2.52/0.82	1.06/0.69 (37%)	7.72/2.96	5.33/1.81 (44%)	7.39/4.45	5.42/3.25 (36%)

TABLE I: Performance comparison of different combinations of planners and PR2 benchmarks (in seconds). We show both the average time and the standard deviation (average time/standard deviation). 'Fail' means all the queries cannot find a collision-free path within 1,000 seconds. The percentage in the brackets shows the speedup obtained while using our modified planners. The benchmarks (c) and (d) contain narrow passages.

	PRM	I-PRM	lazyPRM	I-lazyPRM	RRT	I-RRT	RRT*	I-RRT*
apartment	5.25/0.81	2.54/0.65 (106%)	2.8/0.32	1.9/0.23 (47%)	0.09/0.12	0.10/0.11 (-10%)	0.22/0.16	0.23/0.14 (5%)
cubicles	3.92/0.66	2.44/0.51 (60%)	1.62/0.57	1.37/0.43 (19%)	0.89/0.52	0.87/0.44 (2%)	1.95/0.83	1.83/0.91 (7%)
easy	7.90/1.02	5.19/0.86 (52%)	3.03/1.12	2.01/0.94 (50%)	0.13/0.59	0.15/0.55 (-13%)	0.26/0.17	0.27/0.15 (-4%)
flange	fail	fail	fail	fail	48.47/25.43	25.6/11.17 (88%)	46.07/20.52	26.9/11.67 (73%)
torus	31.52/4.3	23.3/3.5 (39%)	4.16/0.91	2.75/0.88 (51%)	3.95/1.12	2.7/0.93 (46%)	6.01/2.1	4.23/1.65 (42%)

TABLE II: Performance comparison of different combinations of planners and rigid body benchmarks (in seconds). We show both the average time and the standard deviation (average time/standard deviation). 'Fail' means all the queries cannot find a collision-free path within 1,000 seconds. The percentage in the brackets shows the speedup obtained while using our modified planners. Both flange and torus benchmarks contain narrow passages.
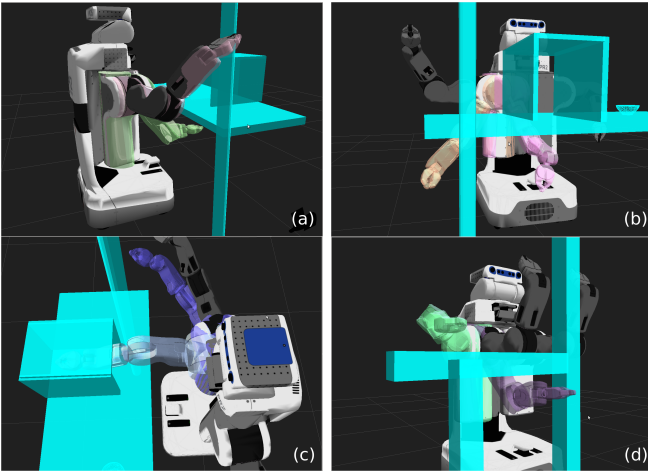

Fig. 6: PR2 planning benchmarks: robot arms with different colors show the initial and goal configurations. The first three benchmarks are of the same environment, but the robot's arm is in different places: (a) moves arm from under desk to above desk; (b) moves arm from under desk to another position under desk; and (c) moves arm from inside the box to outside the box. In the final benchmark, the robot tries to move arm from under a shelf to above it. The difficulty order of the four benchmarks is (c) > (d) > (b) > (a). These planning problems are for PR2's 7-DOF robot arm.
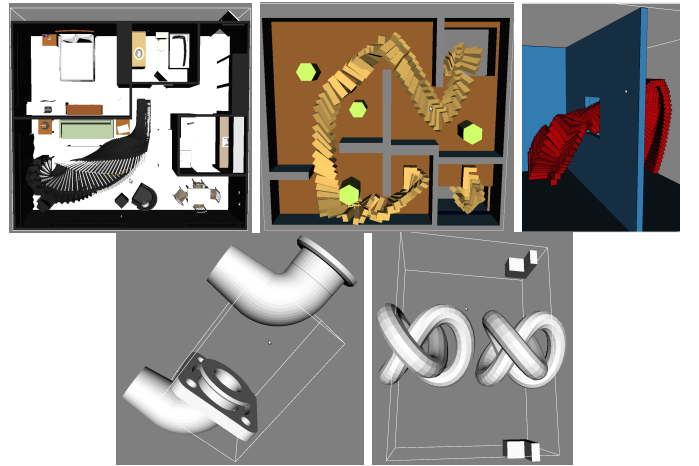

Fig. 7: Rigid body planning benchmarks: from left to right, apartment, cubicles, easy, flange and torus. Apartment benchmark tries to move the piano to the hallway near the door entrance; in cubicles benchmark, the robot moves through a simple office-like environment where the robot needs to fly through the basement; both flange and torus benchmarks contain narrow passages. These problems are with 6 DOFs.

that are well-explored, we should avoid inserting collision results into the dataset in order to limit the dataset size. For instance, we can choose to only add samples that have large information gain [38]. In addition, the probabilistic collision query can also be used for developing learning algorithms for motion planning [7]. Since prior collision results are stored in hash tables, we can efficiently update this data without high overhead. Therefore, we may be able to extend our approach to improve the performance of motion planning algorithms in dynamic environments. In particular, we divide the workspace into a set of grids. For each grid cell, we can maintain two sets: the set of obstacles intersecting with the cell, and the set of configurations in the database $\mathcal{D}$ that will make the robot collide with the cell. The maintenance of both sets can be implemented efficiently with the update operation on a spatial hash table. Given a moving obstacle, we can first locate the set of grid cells that overlap with the obstacle's swept volume (between two successive time instances), and compute the set of configurations whose collision status may change due to this movement. These two queries can also be accelerated by leveraging the hash table data structure. Finally, we can update the collision status for these samples in $\mathcal{D}$ using exact collision detection algorithm. This update operation can be performed efficiently since our database is implemented as a hash table. It will be useful to evaluate this approach in dynamic scenes.
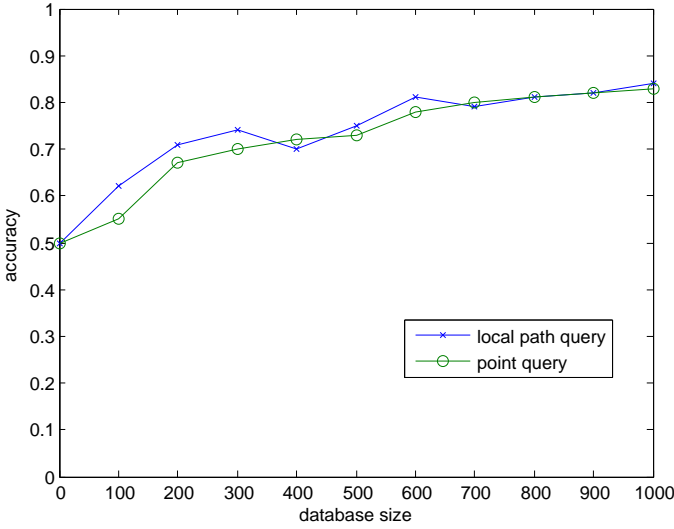
Fig. 8: The accuracy result for benchmark shown in Figure 6(a). For databases of different sizes, we compute the average accuracy on 100 point queries or local path queries. If an ambiguity rejection or a distance rejection happens for a query, we measure its accuracy as 0.5 because any estimate is no better than random guess, and we in fact perform exact collision checking on such samples. This is why the accuracy is 0.5 when the database is empty.
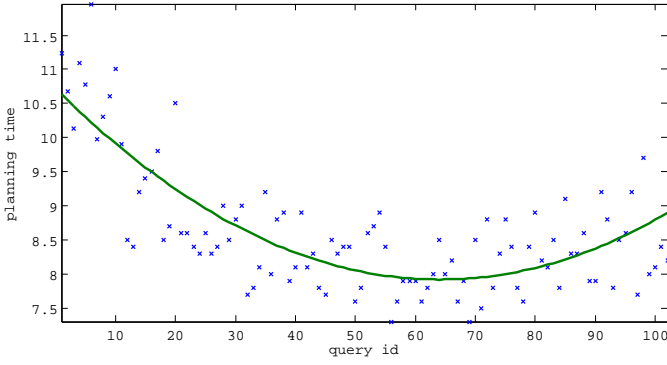

Fig. 9: The time taken by I-PRM when it runs more than 100 times on the benchmark shown in Figure 6(a). The planning time of a single query first decreases and then increases. The best acceleration acquired is $12.78/7.5 = 70\%$, larger than the 32% in Table I.

Appendix A
Proof of Theorem 2

Using the result of random projections, for any point $\mathbf{x}$ and any line $l(\mathbf{v}, \mathbf{a})$, we have

$$\mathbb{P}[h^{\mathbf{u}}(-\hat{V}(\mathbf{v}, \mathbf{a})) = h^{\mathbf{u}}(\hat{V}(\mathbf{x}))]$$
$$= 1 - \frac{1}{\pi} \cos^{-1}\left(\frac{-\hat{V}(\mathbf{v}, \mathbf{a})^T \hat{V}(\mathbf{x})}{\|\hat{V}(\mathbf{v}, \mathbf{a})\| \|\hat{V}(\mathbf{x})\|}\right),$$

where $\mathbf{u} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. According to our embedding design in Section IV-A1, we have $\hat{V}(\mathbf{v}, \mathbf{a})^T \hat{V}(\mathbf{x}) = -\operatorname{dist}^2(\mathbf{x}, l)$, $\|\hat{V}(\mathbf{x})\| = c$ and $\|\hat{V}(\mathbf{v}, \mathbf{a})\| = \sqrt{d - 2 + (\operatorname{dist}^2(\mathbf{0}, l) + 1)^2}$. Let $C = \|\hat{V}(\mathbf{x})\| \|\hat{V}(\mathbf{v}, \mathbf{a})\|$, which is a constant value independent with database points and only depends on the query, then we have $\mathbb{P}[\operatorname{sgn}(\mathbf{u}^T(-\hat{V}(\mathbf{v}, \mathbf{a}))) = \operatorname{sgn}(\mathbf{u}^T \hat{V}(\mathbf{x}))] = 1 - \frac{1}{\pi} \cos^{-1}\left(\frac{-\operatorname{dist}^2(\mathbf{x}, l)}{C}\right) = \frac{1}{\pi} \cos^{-1}\left(\frac{\operatorname{dist}^2(\mathbf{x}, l)}{C}\right)$.
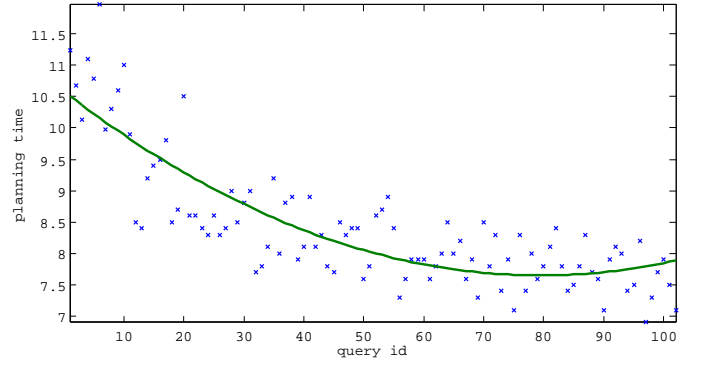

Fig. 10: The time taken by I-PRM when it runs more than 100 times on the benchmark shown in Figure 6(a). In this experiment, we stop adding a collision query result into the database while the local sample density around the query is larger than a given threshold. As the database size does not change much after about the 60-th query, the system does not suffer from the re-increase of the planning time.

Hence, when $\operatorname{dist}(\mathbf{x}, l) \leq r$, we have

$$\mathbb{P}[h^{\mathbf{u}}(-\hat{V}(\mathbf{v}, \mathbf{a})) = h^{\mathbf{u}}(\hat{V}(\mathbf{x}))] \geq \frac{1}{\pi} \cos^{-1}\left(\frac{r^2}{C}\right) = p_1,$$

and when $\operatorname{dist}(\mathbf{x}, l) \geq r(1 + \epsilon)$, we have

$$\mathbb{P}[h^{\mathbf{u}}(-\hat{V}(\mathbf{v}, \mathbf{a})) = h^{\mathbf{u}}(\hat{V}(\mathbf{x}))] \leq \frac{1}{\pi} \cos^{-1}\left(\frac{r^2(1 + \epsilon)^2}{C}\right) = p_2.$$

Then we can estimate the bound for $\rho = \frac{\log p_1}{\log p_2}$ as follows:

$$\rho = \frac{\log\left(\frac{1}{\pi} \cos^{-1}\left(\frac{r^2}{C}\right)\right)}{\log\left(\frac{1}{\pi} \cos^{-1}\left(\frac{r^2(1+\epsilon)^2}{C}\right)\right)} \approx \frac{\log\left(\frac{1}{2} - \frac{r^2}{\pi C}\right)}{\log\left(\frac{1}{2} - \frac{r^2(1+\epsilon)^2}{\pi C}\right)}$$

$$= \frac{-\log 2 + \log\left(1 - \frac{2r^2}{\pi C}\right)}{-\log 2 + \log\left(1 - \frac{2r^2(1+\epsilon)^2}{\pi C}\right)} \approx \frac{-\log 2 - \frac{2r^2}{\pi C}}{-\log 2 - \frac{2r^2(1+\epsilon)^2}{\pi C}},$$

where the first and second approximations are due to the Taylor series of $\cos^{-1}(x)$ and $\log(1-x)$. As a result, we have $\frac{1}{(1+\epsilon)^2} \leq \rho \leq 1$ and $\rho \approx \frac{1}{(1+\epsilon)^2}$ if $\frac{r^2}{C}$ is large enough. ∎

Remark According to the proof above, as the computational complexity of LSH is $\mathcal{O}(N^\rho)$ for a given approximation level ϵ, the line-point k-NN has sub-linear complexity only if $\frac{r^2}{C}$ is large enough. As $C = c \cdot \sqrt{d - 2 + (\operatorname{dist}^2(\mathbf{0}, l) + 1)^2}$, this requires $\operatorname{dist}^2(\mathbf{0}, l)$ to be small. As a result, the computational complexity is query-dependent, which is caused by the affine property of the line. In fact, such undesired property will disappear if we constrain all lines passing through the origin, because $\operatorname{dist}(\mathbf{0}, l) = 0$ for all l. To avoid the performance reduction for lines far from the origin, we uniformly sample several points within the space as the origins of different coordinate systems. Next, we perform line-point hashing for dataset points relative to each coordinate system and store them in different hash tables. Given a line query, we also perform a similar hashing process. As the line is likely to have only a small distance to the origin in one of the coordinate systems, the computational complexity can nearly be $\mathcal{O}(N^{\frac{1}{(1+\epsilon)^2}})$.

APPENDIX B
PROOF OF THEOREM 3

Using the result in [31], for any point $\mathbf{x}$ and any line $l(\mathbf{v}, \mathbf{a})$, we have

$$\mathbb{P}[h^{\mathbf{a},b}(-\hat{V}(\mathbf{v},\mathbf{a})) = h^{\mathbf{a},b}(\hat{V}(\mathbf{x}))]$$
$$= f(\frac{W}{\|\hat{V}(\mathbf{v},\mathbf{a}) + \hat{V}(\mathbf{x})\|}). \tag{15}$$

According to our embedding design in Section IV-A1, $\|\hat{V}(\mathbf{v},\mathbf{a}) + \hat{V}(\mathbf{x})\|^2 = 2\,\mathrm{dist}(\mathbf{x},l) + c^2 + d - 2 + (\mathrm{dist}^2(\mathbf{0},l) + 1)^2$ and we let $C = c^2 + d - 2 + (\mathrm{dist}^2(\mathbf{0},l) + 1)^2$. Then, if $\mathrm{dist}(\mathbf{x},l) \le r$, there is $\|\hat{V}(\mathbf{v},\mathbf{a}) + \hat{V}(\mathbf{x})\| \le \sqrt{2r^2 + C}$; if $\mathrm{dist}(\mathbf{x},l) \ge r(1 + \epsilon)$, there is $\|\hat{V}(\mathbf{v},\mathbf{a}) + \hat{V}(\mathbf{x})\| \ge \sqrt{2r^2(1+\epsilon)^2 + C}$. By putting these bounds into Equation 15, we can obtain p_1 and p_2. Using the result in [31], we have $\frac{1}{1+\epsilon} \le \rho = \sqrt{\frac{2r^2 + C}{2r^2(1+\epsilon)^2 + C}} \le 1$. ∎

Remark As in the case of hamming hashing in Theorem 2, for p-stable hashing we also require small $\mathrm{dist}(\mathbf{0},l)$ to obtain sub-linear complexity, which can be realized by using multiple coordinate systems with origins uniformly sampled in the problem space.

APPENDIX C
PROOF OF THEOREM 4

The proof is in fact only a simple adaption of the proof of Theorem 1 for point-point k-NN presented in [39].

Let $\mathbf{x}^*$ be a point such that $\mathrm{dist}(\mathbf{x}^*,l) \le r$, then for any j, $\mathbb{P}[g_j(\mathbf{x}^*) = g_j(l)] \ge p_1^M = p_1^{\log_{1/p_2} N} = N^{-\rho}$. Hence, $\mathbb{P}[\forall j, g_j(\mathbf{x}^*) \ne g_j(l)] \le (1 - N^\rho)^L = (1 - N^\rho)^{N^\rho} \le 1/e$. Thus, $g_j(\mathbf{x}^*) = g_j(l)$ holds for some $1 \le j \le L$ with probability at least $1 - 1/e$. We denote this property as P_1.

Next, consider set of points whose distances from l are at least $r(1 + \epsilon)$, and which have the same hash code with query l under hash function g_j. We denote the set as $S_j = \{\mathbf{x} \mid \mathrm{dist}(\mathbf{x},l) > r(1 + \epsilon) \text{ and } g_j(\mathbf{x}) = g_j(l)\}$ and let $S = \cup S_j$. Note that the probability for one point in the dataset belonging to S_j is $\mathbb{P}[\mathrm{dist}(\mathbf{x},l) > r(1+\epsilon) \text{ and } g_j(\mathbf{x}) = g_j(l)] = \mathbb{P}[\mathrm{dist}(\mathbf{x},l) > r(1+\epsilon)]\mathbb{P}[g_j(\mathbf{x}) = g_j(l) \mid \mathrm{dist}(\mathbf{x},l) > r(1 + \epsilon)] \le 1 \cdot p_2^M = p_2^{\log_{1/p_2} N} = 1/N$. As a result, $\mathbb{E}(|S_j|) \le 1/N \cdot N = 1$ and $\mathbb{E}(|S|) \le \sum_{j=1}^L \mathbb{E}(|S_j|) \le L$. By Markov's inequality, $\mathbb{P}[|S| > cL] \le \frac{\mathbb{E}(|S|)}{cL} \le \frac{1}{c}$. As a result, $|S| < 2l$ with probability at least $1/2$. As S denotes the data items that the algorithm needs to search but are not valid k-NN results, this property means that the search time is bounded by $|S| = \mathcal{O}(L = N^\rho)$. We denote this property as P_2.

Combining the above two properties, we can find that the probability to obtain at least one point $\mathbf{x}^*$ with $\mathrm{dist}(\mathbf{x}^*,l) \le r$ in bounded retrieval time $\mathcal{O}(N^\rho)$ (i.e., $|S| = 2L$) is $1 - ((1 - \mathbb{P}[P_1]) + (1 - \mathbb{P}[P_2])) \ge 1/2 - 1/e$. ∎

REFERENCES

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

[2] J. Kuffner and S. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *Proceedings of IEEE International Conference on Robotics and Automation*, 2000, pp. 995–1001.

[3] V. Boor, M. Overmars, and A. van der Stappen, "The gaussian sampling strategy for probabilistic roadmap planners," in *Proceedings of IEEE International Conference on Robotics and Automation*, 1999, pp. 1018–1023.

[4] J. Denny and N. M. Amato, "Toggle PRM: Simultaneous mapping of c-free and c-obstacle - a study in 2d," in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011, pp. 2632–2639.

[5] S. Rodriguez, X. Tang, J.-M. Lien, and N. Amato, "An obstacle-based rapidly-exploring random tree," in *Proceedings of IEEE International Conference on Robotics and Automation*, 2006, pp. 895–900.

[6] Z. Sun, D. Hsu, T. Jiang, H. Kurniawati, and J. H. Reif, "Narrow passage sampling for probabilistic roadmap planners," *IEEE Transactions on Robotics*, vol. 21, no. 6, pp. 1105–1115, 2005.

[7] J. Pan, S. Chitta, and D. Manocha, "Faster sample-based motion planning using instance-based learning," in *Workshop on the Algorithmic Foundations of Robotics*, 2012.

[8] N. M. Amato, O. B. Bayazit, L. K. Dale, C. Jones, and D. Vallejo, "OBPRM: an obstacle-based prm for 3d workspaces," in *Proceedings of Workshop on the Algorithmic Foundations of Robotics on Robotics*, 1998, pp. 155–168.

[9] D. Hsu, L. E. Kavraki, J.-C. Latombe, R. Motwani, and S. Sorkin, "On finding narrow passages with probabilistic roadmap planners," in *Proceedings of Workshop on the Algorithmic Foundations of Robotics on Robotics*, 1998, pp. 141–153.

[10] L. Zhang and D. Manocha, "An efficient retraction-based rrt planner," in *Proceedings of International Conference on Robotics and Automation*, 2008, pp. 3743–3750.

[11] L. Jaillet, A. Yershova, S. La Valle, and T. Simeon, "Adaptive tuning of the sampling domain for dynamic-domain RRTs," in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2005, pp. 2851–2856.

[12] A. Yershova, L. Jaillet, T. Simeon, and S. LaValle, "Dynamic-domain RRTs: Efficient exploration by controlling the sampling domain," in *Proceedings of IEEE International Conference on Robotics and Automation*, 2005, pp. 3856–3861.

[13] B. Burns and O. Brock, "Toward optimal configuration space sampling," in *Proceedings of Robotics: Science and Systems*, 2005.

[14] R. A. Knepper and M. T. Mason, "Real-time informed path sampling for motion planning search," *International Journal of Robotics Research*, vol. 31, no. 11, pp. 1231–1250, 2012.

[15] B. Burns and O. Brock, "Sampling-based motion planning using predictive models," in *Proceedings of IEEE International Conference on Robotics and Automation*, 2005, pp. 3120–3125.

[16] J. Pan, S. Chitta, and D. Manocha, "Probabilistic collision detection between noisy point clouds using robust classification," in *International Symposium on Robotics Research*, 2011.

[17] J. Pan, I. Sucan, S. Chitta, and D. Manocha, "Real-time collision detection and distance computation on point cloud sensor data," in *IEEE International Conference on Robotics and Automation*, 2013, pp. 3593–3599.

[18] N. Jetchev and M. Toussaint, "Trajectory prediction in cluttered voxel environments," in *Proceedings of International Conference on Robotics and Automation*, 2010, pp. 2523–2528.

[19] X. Jiang and M. Kallmann, "Learning humanoid reaching tasks in dynamic environments," in *Proceedings of International Conference on Intelligent Robots and Systems*, 2007, pp. 1148–1153.

[20] D. Berenson, P. Abbeel, and K. Goldberg, "A robot path planning framework that learns from experience," in *Proceedings of International Conference on Robotics and Automation*, 2012, pp. 3671–3678.

[21] M. Phillips, B. Cohen, S. Chitta, and M. Likhachev, "E-Graphs: Bootstrapping planning with experience graphs," in *Proceedings of Robotics: Science and Systems*, 2012.

[22] M. Stolle and C. G. Atkeson, "Policies based on trajectory libraries," in *Proceedings of International Conference on Robotics and Automation*, 2006, pp. 3344–3349.

[23] M. S. Branicky, R. A. Knepper, and J. J. Kuffner, "Path and trajectory diversity: Theory and algorithms," in *Proceedings of International Conference on Robotics and Automation*, 2008, pp. 1359–1364.

[24] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2005.

[25] A. Chakrabarti and O. Regev, "An optimal randomised cell probe lower bound for approximate nearest neighbour searching," in *Proceedings of*

IEEE Symposium on Foundations of Computer Science, 2004, pp. 473–482.

[26] A. Andoni, P. Indyk, R. Krauthgamer, and H. L. Nguyen, "Approximate line nearest neighbor in high dimensions," in *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, 2009, pp. 293–301.

[27] P. Jain, S. Vijayanarasimhan, and K. Grauman, "Hashing hyperplane queries to near points with applications to large-scale active learning," in *Neural Information Processing Systems*, 2010.

[28] R. Basri, T. Hassner, and L. Zelnik-Manor, "Approximate nearest subspace search," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 33, no. 2, pp. 266–278, Feb. 2011.

[29] J. Pan, C. Lauterbach, and D. Manocha, "Efficient nearest-neighbor computation for gpu-based motion planning," in *Proceedings of International Conference on Intelligient Robots and Systems*, 2010, pp. 2243–2248.

[30] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Communications of the ACM*, vol. 51, no. 1, pp. 117–122, Jan. 2008.

[31] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of Symposium on Computational Geometry*, 2004, pp. 253–262.

[32] N. Linial, E. London, and Y. Rabinovich, "The geometry of graphs and some of its algorithmic applications," *Combinatorica*, vol. 15, no. 2, pp. 215–245, 1995.

[33] B. Dubuisson and M. Masson, "A statistical decision rule with incomplete knowledge about classes," *Pattern Recognition*, vol. 26, no. 1, pp. 155–165, 1993.

[34] P. Li, T. J. Hastie, and K. W. Church, "Very sparse random projections," in *Proceedings of International Conference on Knowledge Discovery and Data Mining*, 2006, pp. 287–296.

[35] D. A. Cohn, Z. Ghahramani, and M. I. Jordan, "Active learning with statistical models," *Journal of Artificial Intelligence Research*, vol. 4, pp. 129–145, 1996.

[36] D. Hsu, J.-C. Latombe, and R. Motwani, "Path planning in expansive configuration spaces," in *IEEE International Conference on Robotics and Automation*, vol. 3, 1997, pp. 2719–2726 vol.3.

[37] J. Pan, S. Chitta, and D. Manocha, "FCL: A general purpose library for collision and proximity queries," in *Proceedings of International Conference on Robotics and Automation*, 2012, pp. 3859–3866.

[38] S. M. Persson and I. Sharf, "Sampling-based A* algorithm for robot path-planning," *International Journal of Robotics Research*, vol. 33, no. 13, pp. 1683–1708, 2014.

[39] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of International Conference on Very Large Data Bases*, 1999, pp. 518–529.