Efficient high-quality motion planning by fast all-pairs r-nearest-neighbors

Michal Kleinbort, Oren Salzman and Dan Halperin*

***Abstract*— Sampling-based motion-planning algorithms typically rely on nearest-neighbor (NN) queries when constructing a roadmap. Recent results suggest that in various settings NN queries may be the computational bottleneck of such algorithms. Moreover, in several *asymptotically-optimal* algorithms these NN queries are of a specific form: Given a set of points and a radius r report all pairs of points whose distance is at most r. This calls for an application-specific NN data structure tailored to efficiently answering this type of queries. Randomly transformed grids (RTG) were recently proposed by Aiger et al. [1] as a tool to answer such queries and have been shown to outperform common implementations of NN data structures in this context. In this work we employ RTG for sampling-based motion-planning algorithms and describe an efficient implementation of the approach. We show that for motion-planning, RTG allow for faster convergence to high-quality solutions when compared with existing NN data structures. Additionally, RTG enable significantly shorter construction times for batched-PRM variants; specifically, we demonstrate a speedup by a factor of two to three for some scenarios.**

I. Introduction and Related work

Given a robot moving in an environment cluttered with obstacles, motion-planning (MP) algorithms are used to efficiently plan a path for the robot, while avoiding collision with the obstacles [2]. A common approach is to use *sampling-based* algorithms, which abstract the robot as a point in a high-dimensional space called the *configuration space* (C-space) and plan a path in this space. A point, or a configuration, in the C-space represents a placement of the robot that is either collision-free or not, subdividing the C-space $\mathcal{X}$ into the sets $\mathcal{X}_{\text{free}}$ and $\mathcal{X}_{\text{forb}}$, respectively. The structure of the C-space is then studied by constructing a graph, called a *roadmap*, that approximates the connectivity of $\mathcal{X}_{\text{free}}$. The nodes of the graph are collision-free configurations sampled at random. Two (nearby) nodes are connected by an edge if the straight line connecting their configurations is collision-free as well.

Sampling-based MP algorithms are implemented using two primitive operations: *Collision detection* (CD), which is used to asses if a configuration is collision-free or not, and *Nearest neighbor* (NN) search, which is used to efficiently return the neighbor (or neighbors) of a given configuration. The CD operation is also used to test if the straight line connecting two configurations lies in $\mathcal{X}_{\text{free}}$—a procedure referred to as *local planning*. While in theory the cost of NN exceeds that of local planning, in practice it is the latter that is the main computational bottleneck in sampling-based MP algorithms [2].

However, recent results (e.g. [3], [4], [5]) suggest that this may not always be the case. By carefully replacing many expensive calls to the local planner with NN queries, one may reduce the running time of different sampling-based MP algorithms. As a result, the computational overhead of NN queries plays a significant role in the running time of these algorithms. Moreover, existing algorithms that ensure *asymptotic optimality*[1] such as PRM* [6], FMT* [7] and MPLB [4] or *asymptotic near-optimality*[2] (such as ANO-MPLB [4]) can make use of *all-pairs r-nearest-neighbors* queries. That is, given a set P of n points and a radius $r = r(n)$ report all pairs of points $p, q \in P$ such that the distance between p and q is at most r. This calls for application-specific NN data structures tailored to efficiently answering this type of queries. Many implementations of efficient NN data structures exist, e.g., ANN [8], FLANN [9], and E2LSH [10]. However, none of the above methods is tailored for these very specific NN queries that arise in the context of these motion-planning algorithms.

Contribution and paper organization. This paper adopts Randomly Transformed Grids (RTG), an algorithm by Aiger et al. [1] for finding all-pairs r-nearest-neighbors, to sampling-based MP algorithms. We begin by identifying which algorithms can make use of all-pairs r-nearest-neighbors and review them in Section II. Specifically, we discuss the subtleties of using them in an anytime mode versus a batch mode. After an overview of existing NN data structures we present in Section III the RTG algorithm together with a description of our efficient implementation (which will be publicly available, together with additional experimental results in our web-page[3]). We then proceed with a series of experimental results comparing our implementation with different state-of-the-art implementations of NN data structures and show the speedup that is obtained by using our RTG implementation for all-pairs r-nearest-neighbors queries. To test the affect of RTG in MP algorithms, we present a series of simulations demonstrating that RTG allows to speed up the

* Blavatnik School of Computer Science, Tel-Aviv University, Israel

This work has been supported in part by the Israel Science Foundation (grant no. 1102/11), by the German-Israeli Foundation (grant no. 1150-82.6/2011), and by the Hermann Minkowski–Minerva Center for Geometry at Tel Aviv University.

[1] An algorithm is said to be asymptotically optimal, if the cost of the solution produced by the algorithm asymptotically approaches the cost of the optimal solution. The notion of cost depends on the problem at hand and may be, e.g., path length, energy consumption along the path or minimal distance from the obstacles.

[2] An algorithm is said to be asymptotically near-optimal, if given an approximation factor ε, the cost of the solution produced by the algorithm approaches a cost within a factor of $(1+\varepsilon)$ of the optimal solution.

[3] `http://acg.cs.tau.ac.il/projects/rtg`

construction time of PRM-type roadmaps, the time to obtain an initial solution or to converge to high-quality solutions in complex scenarios. For example, the construction time of PRM-type roadmaps in certain scenarios is reduced by a factor of between two and three. We conclude in Section V with a discussion of the current limitations of our approach together with suggestions for possible future work.

II. Preliminaries

We first review several sampling-based MP algorithms that can rely on NN queries of type all-pairs r-nearest-neighbors. We then continue to discuss existing NN data structures.

A. Sampling-based motion-planning algorithms

Throughout this subsection we use the following procedures, which are standard procedures used in sampling-base MP algorithms. $\texttt{sample\_free}(n)$ is a procedure returning n random free configurations. $\texttt{nearest\_neighbor}(x, V)$ and $r\texttt{-nearest\_neighbors}(x, V, r)$ return the nearest neighbor and all nearest neighbors in a ball of radius r of x within the set V, respectively. Let $\texttt{steer}(x, y)$ return a configuration z that is closer to y than x is. The procedure $\texttt{collision\_free}(x, y)$ tests whether the straight-line segment connecting x and y is contained in $\mathcal{X}_{\text{free}}$, and $\texttt{dist}(x, y)$ returns the Euclidean distance of the straight-line path connecting x and y. Let us denote by $\texttt{cost}_{\mathcal{G}}(x)$ the minimal cost[4] of reaching a node x from x_{init} using a roadmap $\mathcal{G}$.

We begin by discussing the subtle difference between batch and anytime MP algorithms. We refer to an algorithm as a *batch* algorithm if it processes a predefined number of samples n in one go. An algorithm is said to be *anytime* if it refines its solution as time progresses and may be run for any given amount of time. Observe that efficiently computing all-pairs r-nearest-neighbors is intrinsically a *batch* operation. We briefly describe a scheme to easily modify a batch algorithm into an anytime one (see, e.g., [11]). First, run the algorithm on an initial (small) set of n samples. Then, as long as time permits, double n and re-run the algorithm using the larger n. This way we obtain anytime algorithms, which rely on all-pairs r-nearest-neighbors—this in turn makes them amenable to the optimization that we propose in this paper. We note that between iterations, additional optimizations such as pruning, informed sampling [12], using lower bounds [4] or relaxing optimality [13] may be applied.

Arguably, the best-known MP algorithm that makes use of all-pairs r-nearest-neighbors is PRM* [6]. PRM*, outlined in Alg. 1, is a multi-query, batch, asymptotically-optimal algorithm that maintains a graph data structure as its roadmap. It samples n collision-free configurations which are the vertices of the roadmap (line 1). Two configurations are connected by an edge if their distance is less than $r(n)$ and if the straight-line connecting them is collision-free (lines 2-5). Specifically,

[4]In this paper, unless stated otherwise, we use Euclidean distance as the cost function.

Algorithm 1 PRM* (n)

1: $V \leftarrow \{x_{\text{init}}\} \cup \texttt{sample\_free}(n)$; $E \leftarrow \emptyset$; $\mathcal{G} \leftarrow (V, E)$
2: **for all** $(x, y \in V)$ **do**
3: **if** $(\texttt{dist}(x, y) \leq r(n))$ **then**
4: **if** $(\texttt{collision\_free}(x, y))$ **then**
5: $E \leftarrow E \cup (x, y)$

the radius used [6] is

$$r_{\text{PRM}^*} = 2\left[\left(1+\frac{1}{d}\right)\cdot\left(\frac{\mu(\mathcal{X}_{\text{free}})}{\zeta_d}\right)\cdot\left(\frac{\log n}{n}\right)\right]^{1/d}, \quad (1)$$

where d is the dimension, $\mu(\mathcal{X}_{\text{free}})$ is the volume of the free space and ζ_d is the volume of a d-dimensional sphere of radius 1.

To reduce the number of calls to the local planner, one can delay local planning to the query phase and test if two neighbors are connected only if they potentially lie on the shortest path to the goal. This lazy approach was originally suggested for the PRM algorithm [14]. We apply this approach to the aforementioned batched PRM* and call it LazyB-PRM*. Somewhat similarly, Luo and Hauser [5] have recently proposed an anytime, *single-query* variant for PRM* called Lazy-PRM* that relies on dynamic shortest path algorithms to efficiently update the roadmap.

Janson and Pavone [7] introduced the Fast Marching Tree algorithm (FMT*). The single-query asymptotically-optimal algorithm, outlined in Alg. 2, maintains a tree as its roadmap. Similarly to PRM*, FMT* samples n collision-free nodes V (line 1). It then builds a minimum-cost spanning tree rooted at the initial configuration by maintaining two sets of nodes H, W such that H is the set of nodes added to the tree that may be expanded and W is the set of nodes not in the tree yet (line 2). It then computes for each node the set of nearest neighbors[5] of radius $r(n)$ (line 4). The algorithm repeats the following process: the node z with the lowest cost-to-come value is chosen from H (line 5 and 17). For each neighbor x of z that is not already in H, the algorithm finds its neighbor $y \in H$ such that the cost-to-come of y added to the distance between y and x is minimal (lines 8-10). If the local path between y and x is free, x is added to H with y as its parent (lines 11-13). At the end of each iteration z is removed from H (line 14). The algorithm runs until a solution is found or there are no more nodes to process. The algorithm, together with its bidirectional variant [15], were shown to converge to an optimal solution faster than PRM* and RRT* [6]. The radius used by the FMT* algorithm [7] is

$$r_{\text{FMT}^*} = 2(1+\eta)\left[\left(\frac{1}{d}\right)\cdot\left(\frac{\mu(\mathcal{X}_{\text{free}})}{\zeta_d}\right)\cdot\left(\frac{\log n}{n}\right)\right]^{1/d}, \quad (2)$$

[5]The nearest-neighbor computation can be delayed and performed only when needed but we present the batched mode of computation to simplify the exposition. The delayed variant makes use of multiple r-nearest neighbors queries while the batched-mode variant makes use of all-pairs r-nearest neighbors queries.

Algorithm 2 FMT* (x_{init}, n)

1: $V \leftarrow \{x_{\text{init}}\} \cup \texttt{sample\_free}(n)$; $E \leftarrow \emptyset$; $\mathcal{T} \leftarrow (V, E)$
2: $W \leftarrow V \setminus \{x_{\text{init}}\}$; $H \leftarrow \{x_{\text{init}}\}$
3: **for all** $v \in V$ **do**
4: $N_v \leftarrow \texttt{nearest\_neighbors}(V \setminus \{v\}, v, r(n))$
5: $z \leftarrow x_{\text{init}}$
6: **while** $z \notin \mathcal{X}_{\text{Goal}}$ **do**
7: $H_{\text{new}} \leftarrow \emptyset$; $X_{\text{near}} \leftarrow W \cap N_z$
8: **for** $x \in X_{\text{near}}$ **do**
9: $Y_{\text{near}} \leftarrow H \cap N_x$
10: $y_{\min} \leftarrow \arg\min_{y \in Y_{\text{near}}} \{\texttt{cost}_{\mathcal{T}}(y) + \texttt{dist}(y, x)\}$
11: **if** $\texttt{collision\_free}(y_{\min}, x)$ **then**
12: $\mathcal{T}.\texttt{parent}(x) \leftarrow y_{\min}$
13: $H_{\text{new}} \leftarrow H_{\text{new}} \cup \{x\}$; $W \leftarrow W \setminus \{x\}$
14: $H \leftarrow (H \cup H_{\text{new}}) \setminus \{z\}$
15: **if** $H = \emptyset$ **then**
16: **return** FAILURE
17: $z \leftarrow \arg\min_{y \in H} \{\texttt{cost}_{\mathcal{T}}(y)\}$
18: **return** PATH

Algorithm 3 RRT* (x_{init}, n)

1: $V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$; $\mathcal{T} \leftarrow (V, E)$
2: **for** $i = 1 \dots n$ **do**
3: $x_{rand} \leftarrow \texttt{sample\_free}()$
4: $x_{nearest} \leftarrow \texttt{nearest\_neighbor}(x_{rand}, V)$
5: $x_{new} \leftarrow \texttt{steer}(x_{nearest}, x_{rand})$
6: **if** $(\texttt{collision\_free}(x_{nearest}, x_{new}))$ **then**
7: $V \leftarrow V \cup \{x_{new}\}$; $\mathcal{T}.\texttt{parent}(x_{new}) \leftarrow x_{nearest}$
8: $X_{near} \leftarrow r\text{-}\texttt{nearest\_neighbors}(x_{new}, V, n)$
9: **for all** (x_{near}, X_{near}) **do**
10: $\texttt{rewire\_RRT}^*(x_{near}, x_{new})$
11: **for all** (x_{near}, X_{near}) **do**
12: $\texttt{rewire\_RRT}^*(x_{new}, x_{near})$

Algorithm 4 $\texttt{rewire\_RRT}^*(x_{potential\_parent}, x_{child})$

1: **if** $(\texttt{collision\_free}(x_{potential\_parent}, x_{child}))$ **then**
2: $c \leftarrow \texttt{dist}(x_{potential\_parent}, x_{child})$
3: **if** $(\texttt{cost}_{\mathcal{T}}(x_{potential\_parent}) + c < \texttt{cost}_{\mathcal{T}}(x_{child}))$ **then**
4: $\mathcal{T}.\texttt{parent}(x_{child}) \leftarrow x_{potential\_parent}$

where $\eta > 0$ is some small constant. Moreover, Janson and Pavone show that PRM* can also use the (smaller) radius defined in Eq. 2 while maintaining its asymptotic optimality.

Recently, we proposed a scheme to compute tight, effective lower bounds on the cost to reach the goal [4]. Incorporating these bounds with the FMT* algorithm, we introduced Motion Planning using Lower Bounds or MPLB. The algorithmic tools used by MPLB cause the weight of collision-detection to be negligible when compared to NN calls. Some of the experimental results suggest that more than 40% of the running time of the algorithm is spent on NN queries. MPLB uses the same radius as FMT* (Eq. 2).

Both FMT* and MPLB perform r-nearest-neighbors queries for a subset of the input points (we call this *multiple r-nearest-neighbors queries)*. On the other hand, the NN data structure we propose to use answers only all-pairs r-nearest-neighbors queries. This can easily be addressed by performing an all-pairs r-nearest-neighbors query once, and storing all such pairs. Multiple r-nearest-neighbors queries are then reduced to querying the stored set of pairs and returning the relevant ones. Clearly, this will only be efficient when the number of multiple r-nearest-neighbors queries is large. As demonstrated in Section IV this is indeed the case.

RRT* [6] is a single-query asymptotically-optimal variant of the RRT algorithm [16]. We first outline the RRT algorithm (lines 1-7 of Alg. 3) and then continue describing the RRT* algorithm together with a variant which we call batched-RRT*. The RRT algorithm maintains a tree as its roadmap. At each iteration a configuration x_{rand} is sampled at random (line 3). Then, $x_{nearest}$, the nearest configuration to x_{rand} in the roadmap is found (line 4) and extended in the direction of x_{rand} to a new configuration x_{new} (line 5). If the path between $x_{nearest}$ and x_{new} is collision-free, then x_{new} is added to the roadmap (lines 6-7).

RRT* follows the same steps as the RRT algorithm but has an additional stage after an edge is added to the tree (lines 8-12 of Alg. 3): a set X_{near} of the r-nearest neighbors of x_{new} is considered and a *rewiring* step (see Alg. 4) occurs twice: first, it is used to find the node $x_{near} \in X_{near}$ which will minimize the cost to reach x_{new}; then, the procedure is used to attempt to minimize the cost to reach every node $x_{near} \in X_{near}$ by considering x_{new} as its parent. We note that RRT* uses two types of NN queries: both nearest-neighbor and multiple r-nearest_neighbors. Moreover, the original formulation of RRT* [6] uses at step i a radius of $r_{RRT*}(i)$ where

$$r_{\text{RRT}^*}(i) = \left[\left(2\left(1 + \frac{1}{d}\right)\right) \cdot \left(\frac{\mu(\mathcal{X}_{\text{free}})}{\zeta_d}\right) \cdot \left(\frac{\log i}{i}\right)\right]^{1/d}. \quad (3)$$

However, to ensure asymptotic optimality, a radius of $r(n)$ may be used at each stage of the algorithm (see proof of Thm. 38 in [6]).

We propose a batch variant of RRT* that stems from the FMT* framework. Instead of using n uniformly sampled configurations, one may use the n nodes constructed by an RRT algorithm and build the FMT*-tree using these nodes. This variant benefits from (i) the fast exploration of the configuration space due to the Voronoi-bias that RRT has and (ii) the efficient construction of the shortest-path spanning tree due to the Dijkstra-like pass that FMT* has. We call this variant batched-RRT*.

We summarize the different algorithms in Table I together with the type of NN queries that they use.

B. Review of existing nearest-neighbor data structures

As nearest-neighbors search is widely used in various domains there exists a wide range of methods allowing efficient

TABLE I
List of algorithms that use r-nearest neighbors queries.

Algorithm	NN queries used	Comments
PRM*	all-pairs r-NN	Multi-query
LazyB-PRM*	all-pairs r-NN	Multi-query
FMT*	multiple r-NN	All-pairs variant exists Single-query
MPLB	multiple r-NN	All-pairs variant exists Single-query, anytime
batched-RRT*	nearest neighbor multiple r-NN	All-pairs variant exists

proximity queries, differing in their space requirement and time complexity. Such methods include the kd-trees [17], [18], geometric near-neighbor access trees (GNAT) [19], locality sensitive hashing (LSH) [20], and others [21].

kd-trees, which work best for rather low dimensions, are often used in motion-planning settings. A kd-tree is a binary tree storing the input points in its leaves, where each node v defines an axis-aligned hyper-rectangle containing the points stored in the subtree rooted at v. Given a query point q, NN search is performed in two phases: the first locates the leaf node with the hyper-rectangle containing q, and the second traverses the tree backwards searching for closer sibling nodes. Given a d-dimensional point set of size n, construction takes $O(dn \log n)$ time. Friedman et al. [18] showed that under mild assumptions the expected time for a single nearest-neighbor query is $O(\log n)$. For r-nearest-neighbors queries, the expected complexity is at least $\Omega(\log n + k)$, where k is the number of reported neighbors (the worst-case bounds are much worse [22]).

Another recursive structure that is frequently used in motion-planning algorithms is GNAT. The input point set is recursively divided into smaller subsets and each subset is then represented using a subtree. Searching the structure is done recursively, while the recursion call continues to child nodes that have not yet been pruned. As claimed in [19], typically only linear space is required and the construction takes $O(dn \log n)$ time.

A common practice for speeding up algorithms that use NN queries but do not require exact results (such as MP algorithms) is to use *approximate* NN queries. Different types of approximate r-nearest-neighbors methods exists: some (e.g., [8]) return with high probability most neighbors of a given query point q, while others return a neighbor within a distance $r(1+\epsilon)$ if a neighbor at distance of at most r from q exists.

Locally sensitive hashing (LSH), presented by Indyk and Motwani [20], is an approximate nearest-neighbor method for d-dimensional point sets. As opposed to many other methods, its $O(dn^{\frac{1}{1+\epsilon}})$ query time does not depend exponentially on the dimension. LSH uses a subset of t hash functions from a family of locally sensitive hash functions for mapping the data into buckets. That is, with high probability two close points will be mapped to the same bucket. Given a query point q, the algorithm maps q using the set of hash functions to a set of buckets and collects all data points that were mapped to these buckets. The required neighbor is then found within this set of collected candidates.

An algorithm for finding all nearest-neighbors in a given fixed d-dimensional point set under the L_∞-metric was originally presented by Lenhof and Smid [23]. The algorithm uses a fixed uniform grid for inspecting pairs of points that lie in the same grid cell or in two adjacent ones. Its time complexity is linear in both the input size and the output size. A simplified variant was later presented by Chan [24].

Algorithm 5 Randomly Shifted Grids (P, r, c, m)

1: **for** $i \in \{1 \dots m\}$ **do**
2: Choose a random shift for a grid of cell size c
3: $U \leftarrow \emptyset$
4: **for all** $p \in P$ **do**
5: Compute the grid cell u that p lies in
6: Associate p to u
7: $U \leftarrow U \cup \{u\}$
8: **for all** $u \in U$ **do**
9: Go over all pairs of points in u and report those of Euclidean distance at most r

III. Randomly Transformed Grids (RTG)

Aiger et al. [1] suggest two simple randomized algorithms for approximately answering all-pairs r-nearest-neighbors queries given a set P of n points in a d-dimensional Euclidean space.

The first algorithm, outlined in Alg. 5, conceptually places a d-dimensional axis-parallel grid of cell size[6] c, which is shifted according to a randomly chosen uniform shift (line 2). This grid defines a partition of the points in P into cells, and each point is associated to the cell u containing it (lines 4-7). For each non-empty grid cell, the distance between every two points associated to the cell is computed. A pair inside a cell is then reported if the computed distance is at most r (lines 8-9). The process is repeated m times to guarantee that, with high probability, most pairs of points at Euclidean distance at most r will be reported. The second algorithm follows the same approach adding a random orientation to each randomly shifted grid.

Assuming a constant-cost implementation of the floor function and of hashing, Aiger et al. [1] show that for appropriate choices of c and m, with high probability, the algorithm reports every pair at distance at most r with time $O((n+k)\log n)$, where k is the number of pairs at distance at most r. Note that the hidden constant depends exponentially on the dimension d. When only randomly shifted grids are used, the constant is roughly $(.484\sqrt{d})^d$. However, when both rotations and translations are used, it is roughly 6.74^d. For input sets lying in a subset of the space that has low doubling dimension, much tighter bounds can be obtained.

[6] The cell size c should be slightly larger than the radius r. Thus, to specify the cell size, one needs to specify a constant, which we call the cell-size factor, $\tilde{c}(r,d) > 1$, such that $c = \tilde{c} \cdot r$.

We next discuss the effect of the algorithm's parameters, outline several tradeoffs, and describe an efficient implementation of the algorithm.

A. The key parameters

RTG requires the user to set the cell size c and the number m of randomly shifted grids. These two parameters have a major impact on the performance of the algorithm. When a very small c or m is used, the set of reported pairs might be small compared to the true number of neighboring pairs. However, when a large value of c is used, the algorithm may output the whole set of true neighbors at the cost of performing several inefficient brute-force searches. Obviously, the more iterations performed (larger m) the better the results are. Therefore, a tradeoff between the running time and the quality of the results exists, and any subtle change to either c or m may affect both measures. Experiments supporting this observation are detailed in Subsec. IV-A.

B. Implementation details

Implementing the algorithm in a naïve manner is straightforward. One has to store the non-empty cells and their associated points. All pairs of points within a cell are examined in a brute force manner, computing the distance of a pair in $O(d)$ time and reporting the pair if relevant.

Note that a certain pair of points may be associated to the same cell in several different grids. Thus, in order to report every neighboring pair only once, an auxiliary data structure for storing the already reported pairs is needed. If such a structure allows efficient query operations as well as efficient insertions, it can be utilized to avoid costly distance computations by filtering out many potential pairs. As a result, the running time of the algorithm can be significantly reduced at the cost of storing the auxiliary pairs structure.

Let $P = \{p_1, \ldots, p_n\}$ be the set of input points. A possible auxiliary structure would be an array of unordered sets, where the ith cell of the array stores all indices $j > i$ such that (p_i, p_j) is a reported pair. The space complexity of such a structure is linear in the size of the output (the number of reported pairs), whereas the expected query time and update time are constant. On the other hand, one can use a different auxiliary structure, whose space complexity is $O(n^2)$, and experience much better query and update times. For instance, a bit array representing a two-dimensional matrix, where the cell (i, j) is set to one if the pair (p_i, p_j) is a reported pair, is a possible structure (notice that only half of the array needs to be stored due to symmetry). It supports constant-time insert and constant-time query operations. Nevertheless, both the space and the query time complexity do not depend on the output size. However, such a solution is restricted to a limited number of input points, depending on the machine on which the query is executed. We refer to this structure as a flattened two-dimensional bit array.

Our C++ implementation supports either auxiliary data structures discussed. The array of unordered sets is implemented as a vector of `boost::unordered_set`-s, while `boost::dynamic_bitset` is used for implementing the flattened bit-array [25]. Although the latter typically exhibits running times that are twice as fast as the former (for dimensions six and above), it is highly non-scalable with respect to memory consumption, thus only applicable to settings where a limited number of samples is required. In the rest of the paper we report on results of the first variant only, namely an array of unordered sets.

As we construct one grid at a time, and since each new grid may introduce new neighboring pairs, the pairs are reported in an unordered manner. Therefore, only all-pairs r-nearest-neighbors queries are supported. Yet, the structures can be extended such that they support multiple r-nearest-neighbors as well. In order to do so, the auxiliary structure should store every pair twice. Then, after finding all pairs, multiple r-nearest-neighbors queries can be easily answered.

We have also implemented the second algorithm proposed by Aiger et al. that adds random orientations to the constructed grids. Our implementation, which uses the Eigen C++ library [26], did not achieve significant improvement in running time and quality of the results, comparing to the first RTG algorithm. Therefore, we do not report on these results here. We leave it for further research to understand the gap between the optimistic theoretical prediction and the effect of orientation in practice.

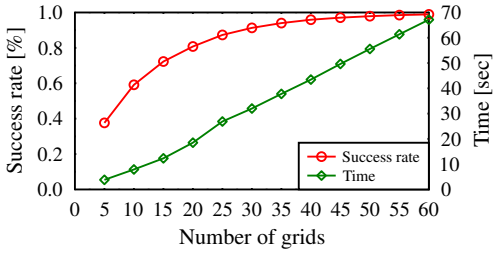
IV. Experimental results

To evaluate our implementation, we first report on a set of experiments aiming both to demonstrate the sensitivity of the RTG algorithm to the parameters used and to determine the better (ideally, optimal) ones. Using these computed parameters, we first compare our implementation with several state-of-the-art implementations of NN data structures and show the speedup that may be obtained by using our RTG implementation for all-pairs r-nearest-neighbors queries. Finally, to test the effect of RTG in MP algorithms, we present a series of simulations that demonstrate that RTG allows to speed up the construction time of a PRM-type roadmap, the time to obtain an initial solution or to converge to high-quality solutions. All experiments were executed on a 2.8GHz Intel Core i7 processor with 8GB of RAM.

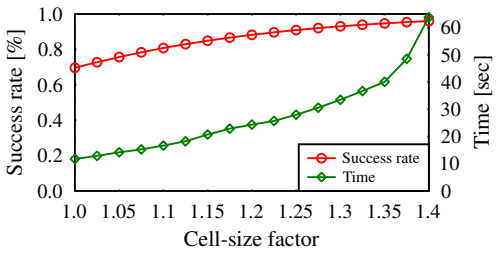
A. Parameter tuning

Recall that given a set of n points in a d-dimensional space and a radius $r = r(n)$ (as in Eq. 2), the RTG algorithm has two parameters that should be set: the cell size c and the number m of grids to construct. As discussed in Sec. III, the algorithm is rather sensitive with respect to either parameter. Therefore, we conducted the following experiments in the unit d-dimensional hypercube: we executed our RTG implementation using increasing values of c and m and measured both the time for running all-pairs r-nearest-neighbor and the success-rate, that is, the ratio between the number of reported pairs and the ground truth. We repeated the experiment for different values of n and d.

Fig. 1a shows that for a given cell-size factor $\tilde{c}$ (recall that $\tilde{c} = \frac{c}{r}$), increasing the number m of grids results in both an increase in the success rate of the algorithm as well



(a)



(b)

Fig. 1. Success rate (red, left axis) and running time (green, right axis) as a function of (a) the number m of grids (for a fixed value of $\tilde{c}$ = 1.1) and (b) the cell-size factor $\tilde{c}$ (for a fixed number of grids $m = 20$). Results are for $n = 102400$ and $d = 9$.

TABLE II

Input parameters (m and $\tilde{c}$) for which the implementation obtained the fastest running times, while reporting at least 98% of neighbors, as a function of the number of points n and the dimension d.

d	3		6		9		12	
n	m	$\tilde{c}$	m	$\tilde{c}$	m	$\tilde{c}$	m	$\tilde{c}$
100	20	1.2	20	1.35	20	1.4	30	1.4
200	20	1.275	20	1.35	20	1.4	30	1.4
400	20	1.2	20	1.35	30	1.3	30	1.4
800	20	1.2	25	1.25	25	1.325	30	1.4
1600	20	1.2	25	1.225	25	1.35	30	1.4
3200	20	1.15	20	1.35	25	1.35	30	1.325
6400	20	1.175	25	1.225	30	1.275	30	1.325
12800	20	1.15	20	1.35	35	1.225	55	1.175
25600	20	1.15	20	1.325	35	1.225	35	1.35
51200	20	1.15	20	1.35	40	1.2	30	1.425
102400	20	1.15	20	1.325	40	1.2	N/A	N/A
204800	20	1.15	20	1.325	N/A	N/A	N/A	N/A

as a linear growth in the running time. Similar behavior was observed for a given m, when $\tilde{c}$ gradually increases (Fig. 1b).

Requiring a success rate of at least 98%, we chose for each d and n the values of m and $\tilde{c}$ (and thus the value of c) that yielded the best running times. Our results are summarized in Table II. We note that similar behavior and results were obtained when running the same experiment on a point set sampled in an environment cluttered with obstacles.

Throughout the next set of experiments, the values for c and m are selected according to the aforementioned table. Note that Aiger et al. [1] use completely different values for c and m, which we found too crude for our setting. The difference lies in the smaller radius used in their experiments, causing the output size to be very small with respect to n. This is not surprising as their data comes from an application of completely different nature.

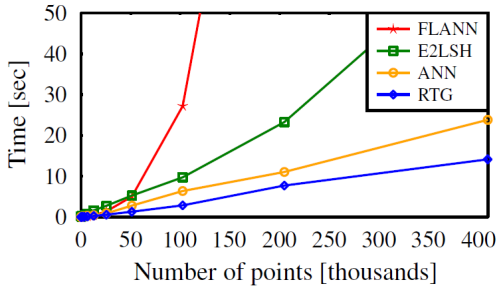


Fig. 2. Comparison between NN methods running all-pairs r-nearest-neighbors for randomly sampled points in the three-dimensional unit hyper-cube.

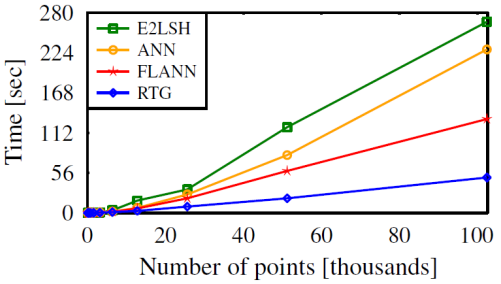


Fig. 3. Comparison between NN methods running all-pairs r-nearest-neighbors for randomly sampled points in the nine-dimensional unit hyper-cube.

B. Comparison with existing NN libraries

We compared our RTG implementation with the following state-of-the-art NN methods: FLANN kd-tree [9], ANN kd-tree [8], and LSH in Euclidean metric spaces (E2LSH) [10]. For ANN we set the bucket size to $\log n$, where n is the number of input points, and allowed an error bound of $\varepsilon = 0.5$ in the query phase. We mark that this had no major effect on the number of reported pairs. For consistency reasons, we used the same error bound in FLANN. For E2LSH we had to empirically estimate the optimal parameters for our point set. Therefore, we used a training set and selected the parameters that minimize the running time on the training set while still reporting a true neighbor with probability at least 0.9. As mentioned, for RTG we chose c and m according to Table II. For each method we measured the time for answering an all-pairs r-nearest-neighbors queries[7] for n random uniform samples from the unit d-dimensional hypercube. The radius $r = r(n)$ was defined as in Eq. 2. We used point sets, of increasing sizes, of dimensions $d = 3, 6, 9,$ and 12.

The results for different dimensions, averaged over ten different runs, are presented in Fig. 2, 3 and 4. Clearly, as the number of samples increases (exactly where NN dominates the running times of MP algorithms) the gap in running time between our RTG implementation and the other methods grows. Similar results (see Fig. 5) were obtained when the environment was cluttered with obstacles.

[7]Recall that for some of the methods finding all pairs requires n single r-nearest-neighbors calls, with each of the n points as an input.

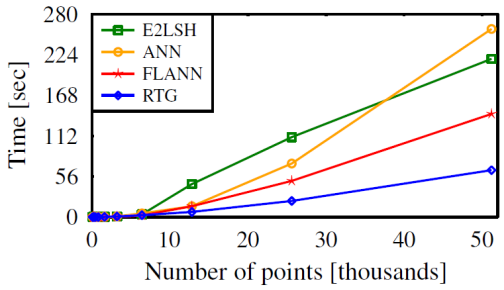


Fig. 4. Comparison between NN methods running all-pairs r-nearest-neighbors for randomly sampled points in the twelve-dimensional unit hyper-cube.

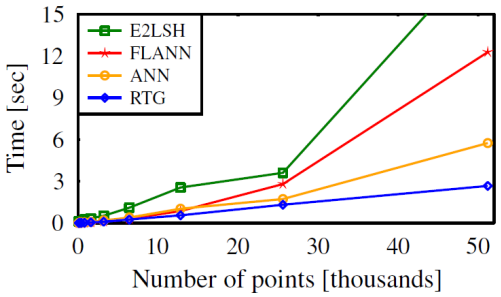
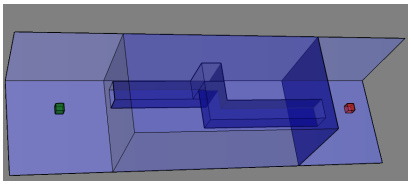
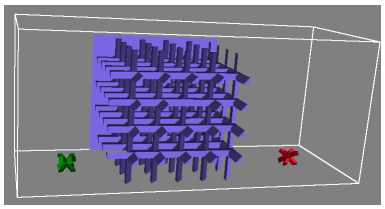


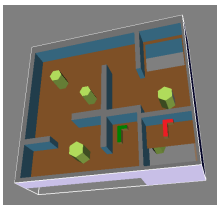
Fig. 5. Comparison between NN methods running all-pairs r-nearest-neighbors for randomly sampled points in an environment cluttered with obstacles (see the Cubicles scenarios, Fig. 6c).



(a) Z-tunnel



(b) 3D Grid



(c) Cubicles

Fig. 6. Scenarios used for MP experiments. When testing the single-query algorithms the green and red robots need to interchange positions.

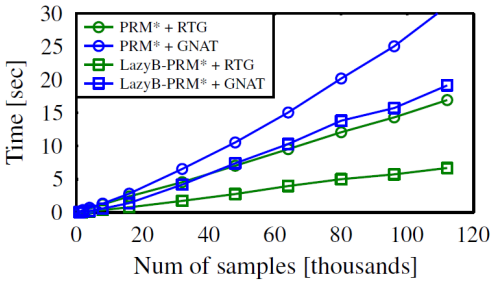


Fig. 7. Roadmap construction time in a three-dimensional Euclidean C-space of a translating robot in the Z-tunnel scenario.

C. RTG in motion-planning algorithms

We integrated our RTG implementation within the OMPL [27] framework, which uses GNAT as its primary NN structure. This allowed us to compare the two NN data structures in different MP algorithms. The scenarios we used are depicted in Fig. 6. Each result is averaged over 50 runs.

We first tested the construction time of the multi-query algorithms PRM* and LazyB-PRM* on the Z-tunnel scenario[8] (Fig. 6a) in a three-dimensional Euclidean C-space of a robot translating in space. Fig. 7 reports on the construction time as a function of the number n of samples in the roadmap. One can clearly see that as n grows, using RTG becomes more advantageous. To test the quality of the roadmap obtained, we performed a query for finding a path from one side of the Z-tunnel to the other (see green and red robots in Fig. 6a). Both roadmaps yielded similar success rates in finding a solution.

Next, we tested the single-query MPLB algorithm on the 3D Grid scenario (Fig. 6b) in a six-dimensional C-space consisting of two translating robots in space. The distance metric we used, which we refer to as MR-metric, computes the sum of the distances that each robot travels. Fig. 8 presents the quality of the solution obtained as a function of time. One can see that RTG allows to find higher quality solutions faster than GNAT. Even though the analysis of Aiger et al. regarding the quality of RTG's results holds only in Euclidean spaces, the MR-metric is more natural in multi-robot settings. We repeated the same test for the Euclidean metric. The results, presented in Fig. 9, demonstrate similar trends to those presented for the MR-metric.

Finally, we report on the success rate of finding a solution in the Cubicles scenario[9] (Fig. 6c). We first performed the experiment for a six-dimensional Euclidean C-space consisting of two translating robots. The results, depicted in Fig. 10, demonstrate that RTG allows to reduce the time to find an initial solution in complex scenarios. Fig. 11 presents similar results using the MR-metric.

V. Discussion and Future Work

Many possible enhancements can be applied to our RTG implementation in order to easily use it in sampling-based MP algorithms. One obvious requirement is to automatically tune the two parameters c and m defining the grid cell size and the number of constructed grids, respectively.

Our RTG implementation is much faster when the flattened bit-array is used for storing the pairs, however this structure has a quadratic space complexity. Thus, a potential solution may introduce a hybrid data structure that for small number of samples uses the bit-array and for large inputs uses the array of unordered sets. Nevertheless, as the array of unordered sets is output sensitive in its memory consumption, when the number of true neighboring pairs is very large,

[8]Scenario based on the Z tunnel scenario by the Parasol MP Group, CS Dept, Texas A&M University `https://parasol.tamu.edu/groups/amatogroup/benchmarks/mp/z_tunnel/`

[9]The Cubicles scenario is provided with the OMPL distribution.

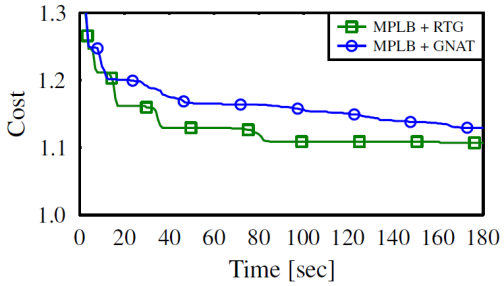


Fig. 8. Solution's cost vs. time in a six-dimensional MR-metric (non-Euclidean) C-space of two translating robots in the 3D Grid scenario. The cost is normalized such that a cost of one denotes the optimal cost that may be obtained.

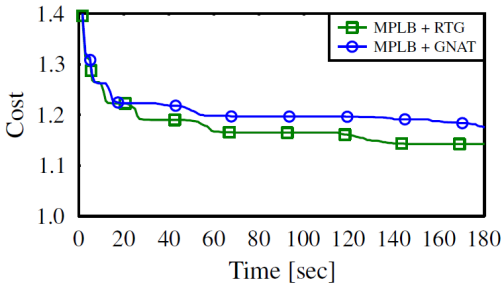


Fig. 9. Solution's cost vs. time in a six-dimensional Euclidean C-space of two translating robots in the 3D Grid scenario. The cost is normalized such that a cost of one denotes the optimal cost that may be obtained.

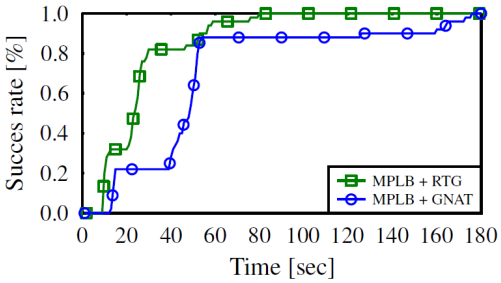


Fig. 10. Success rate to find a solution in a six-dimensional Euclidean C-space of two translating robots in the Cubicles scenario.

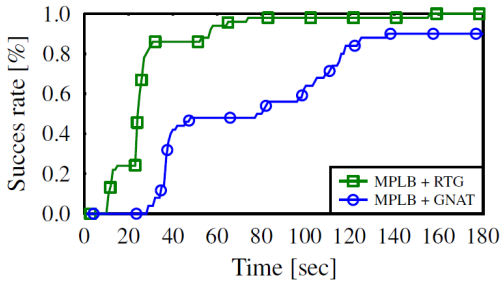


Fig. 11. Success rate to find a solution in a six-dimensional MR-metric (non-Euclidean) C-space of two translating robots in the Cubicles scenario.

cache faults occur and the program slows down drastically. Thus, we seek to have an IO-efficient RTG implementation in order to overcome these limitations.

Similar to the work in [28], one can implement RTG differently such that a single r-nearest-neighbors query, as opposed to all-pairs r-nearest-neighbors, is answered efficiently. Such implementation should store all m constructed grids. At query phase, given a query point q, the m cells containing q are examined, and the set of neighboring points among all potential candidates within the cells is found. For small values of m, this implementation may be efficient.

Another interesting extension is parallelizing the algorithm. A possible approach may be to partition the grid into overlapping portions and run the algorithm on each portion separately. Finally, we wish to devise an RTG variant applicable to non-Euclidean C-spaces, a much needed data structure in asymptotically-optimal sampling-based MP algorithms.

VI. Acknowledgments

We wish to thank Dror Aiger for fruitful discussions regarding efficiently implementing the RTG algorithm.

References

[1] D. Aiger, H. Kaplan, and M. Sharir, "Reporting neighbors in high-dimensional euclidean space," *SIAM J. Comput.*, vol. 43, no. 4, pp. 1363–1395, 2014.

[2] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementation*. MIT Press, June 2005.

[3] J. Bialkowski, S. Karaman, M. W. Otte, and E. Frazzoli, "Efficient collision checking in sampling-based motion planning," in *WAFR*, 2012, pp. 365–380.

[4] O. Salzman and D. Halperin, "Asymptotically near-optimal motion planning using lower bounds on cost," *CoRR*, vol. abs/1403.7714, 2014.

[5] J. Luo and K. Hauser, "An empirical study of optimal motion planning," in *IROS*, 2014, to appear.

[6] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *I. J. Robotic Res.*, vol. 30, no. 7, pp. 846–894, 2011.

[7] L. Janson and M. Pavone, "Fast marching trees: a fast marching sampling-based method for optimal motion planning in many dimensions - extended version," *CoRR*, vol. abs/1306.3532, 2013.

[8] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *J. ACM*, vol. 45, no. 6, pp. 891–923, 1998.

[9] M. Muja and D. G. Lowe, "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration," in *VISSAPP*. INSTICC Press, 2009, pp. 331–340.

[10] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *FOCS*, 2006, pp. 459–468.

[11] W. Wang, D. J. Balkcom, and A. Chakrabarti, "A fast streaming spanner algorithm for incrementally constructing sparse roadmaps," in *IROS*, 2013, pp. 1257–1263.

[12] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Informed RRT*: Optimal incremental path planning focused through an admissible ellipsoidal heuristic," in *IROS*, 2014, to appear.

[13] O. Salzman and D. Halperin, "Asymptotically near-optimal RRT for fast, high-quality, motion planning," in *ICRA*, 2014, pp. 4680–4685.

[14] R. Bohlin and L. E. Kavraki, "Path planning using lazy PRM," in *ICRA*, 2000, pp. 521–528.

[15] J. Starek, E. Schmerling, L. Janson, and M. Pavone, "Bidirectional fast marching trees: An optimal sampling-based algorithm for bidirectional motion planning," *http://web.stanford.edu/ pavone/papers/Starek.ea.pdf*, 2014.

[16] J. J. Kuffner and S. M. LaValle, "RRT-Connect: An efficient approach to single-query path planning," in *ICRA*, 2000, pp. 995–1001.

[17] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, pp. 509–517, Sep. 1975.

[18] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Trans. Math. Softw.*, vol. 3, no. 3, pp. 209–226, Sep. 1977.

[19] S. Brin, "Near neighbor search in large metric spaces," in *VLDB*, 1995, pp. 574–584.

[20] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *STOC*, 1998, pp. 604–613.
[21] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Comput. Surv.*, vol. 30, no. 2, pp. 170–231, 1998.
[22] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Springer-Verlag, 2008.
[23] H. Lenhof and M. H. M. Smid, "Sequential and parallel algorithms for the k closest pairs problem," *Int. J. Comput. Geometry Appl.*, vol. 5, no. 3, pp. 273–288, 1995.
[24] T. M. Chan, "On enumerating and selecting distances," *Int. J. Comput. Geometry Appl.*, vol. 11, no. 3, pp. 291–304, 2001.
[25] B. Schling, *The Boost C++ Libraries*. XML Press, 2011.
[26] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.
[27] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, 2012.
[28] D. Aiger, E. Kokiopoulou, and E. Rivlin, "Random grids: Fast approximate nearest neighbors and range searching for image search," in *ICCV*, 2013, pp. 3471–3478.