2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
November 3-7, 2013. Tokyo, Japan

A fast streaming spanner algorithm for incrementally constructing sparse roadmaps

Weifu Wang Devin Balkcom Amit Chakrabarti

***Abstract*— Sampling-based probabilistic roadmap algorithms such as PRM and PRM* have been shown to be effective at solving certain motion planning problems, but the large graphs generated to express the connectivity and a metric on the configuration space may require much storage space and be expensive to search. Recent work by Marble and Bekris [14], [19] applied *spanner* algorithms to PRM*; these algorithms prune some edges in a dense graph, while guaranteeably maintaining an approximation to the metric. In this paper, we apply (and improve) a state-of-the-art streaming spanner algorithm to prune PRM* roadmaps. The algorithm we present has the main advantage of computational speed; when applied to PRM*, the processing time per vertex is independent of the number of sampled vertices, n, as compared to $O(n \log^2 n \log\log n)$ in [19]. In practice, the algorithm we present prunes a graph with about 20 million edges in less than 20 seconds on a modern desktop computer; compared to the time required for generating such a roadmap, this additional processing time is essentially trivial. In fact, because the combination of this algorithm with PRM* avoids the need for many collision detections, the combination runs several times faster than PRM* alone.**

I. Introduction

Recently, it has been shown that a sampling-based probabilistic roadmap algorithm, PRM* [15], can asymptotically approximate the underlying metric of a configuration space. As might be expected, however, the roadmaps generated contain very many vertices and edges. Work by Marble and Bekris [14], [19], [20] (Incremental Roadmap Spanners, IRS) applied *spanner algorithms* to PRM*; these spanner algorithms relax the metric approximation by a specified multiplicative constant, called the *stretch*, while reducing the number of edges stored in the resulting graph.

In this paper, we introduce a state-of-the-art streaming spanner algorithm from the graph theory community [10] to construct sparse spanner roadmaps incrementally with limited resources (time and space). The advantage of this algorithm over other spanner algorithms is its constant per-edge processing time. We implemented the algorithm and conducted experiments using the *Open Motion Planning Library* (OMPL) [26], but the resulting graphs still contained 60%–90% as many edges as the original; these results are much worse than those reported for the IRS algorithm developed by Marble and Bekris [14], [19].

We have found a variation of the algorithm that prunes essentially as many edges as IRS [14], [19] for sufficiently large stretch or sufficiently large graphs, and we present a proof of the correctness of this variation.

We will refer to the combination of this improved streaming spanner algorithm with PRM* as *SS-PRM**. We believe SS-PRM* to be quite practically useful. The number of neighbors for each vertex is a user-tunable parameter in the original PRM; in a way, SS-PRM* and other spanner algorithms automatically choose this value intelligently and non-uniformly as necessary. In our experiments, we found that compared to PRM with a small fixed number of neighbors, SS-PRM* typically produces much shorter routes in a similar amount of time; in most cases, the median route length of regular PRM is twice as long as SS-PRM*; for details see table V.

Because many edges can be discarded without collision detection, SS-PRM* is several times faster than PRM*, as well as being very much more memory-efficient. And perhaps surprisingly, we have found experimentally that even with large stretch, SS-PRM* does not cause a significant degradation of *average* route length relative to PRM*. For example, even with a stretch of 11, we found the average route length increased by only 10% to 54% in our experiments.

A. Related work

Probabilistic roadmap (PRM) algorithms were developed in the mid-1990s [16]. Shortly after the PRM algorithm was introduced, several other sampling-based algorithms, such as Rapidly-Exploring Random Tree (RRT) [17] and non-holonomic PRM [5], [12], [13] were developed. These sampling-based algorithms became popular almost immediately; they are simple to implement, and can solve many problems that are challenging for deterministic geometric algorithms, such as [7], [9], [18], [25]. Recently, Karaman and Frazzoli [15] proved that a variation on the PRM algorithm, PRM*, asymptotically approximates a metric on the space.

In the graph theory community, over the past decade, many successful spanner algorithms have been developed [3], [8], [22], [23], [4], [24] to sparsify graphs. Extending some of these ideas, Marble *et al.* developed offline [20] as well as online [19], [21] algorithms that could find spanners of roadmaps.

Recently, *streaming* spanner algorithms have been developed [1], [2], [10], [11]. These algorithms process each incoming edge (for example, as the edge is generated by PRM*) only once, while using only a small amount of working memory—always sublinear in the size of the input graph, and typically not much larger than the size of the spanner constructed.

II. Preliminaries

Definition 1: A t-spanner $\hat{G} = (V, \hat{E})$ of an undirected graph $G = (V, E)$ is a subgraph of G (i.e., $\hat{E} \subseteq E$) that satisfies $\hat{d}(u,v) \leq t \times d(u,v)$ for all $u, v \in V$. Here d and $\hat{d}$ are the shortest-path metrics on G and $\hat{G}$ respectively. The factor t is called the *stretch* of the spanner. The definition extends

Weifu Wang, Devin Balkcom, and Amit Chakrabarti are with the Department of Computer Science, Dartmouth College, Hanover, NH 03755 USA. (e-mail: Weifu.Wang.GR@dartmouth.edu, devin@cs.dartmouth.edu, ac@cs.dartmouth.edu).

naturally to weighted graphs (using weighted shortest-path metrics), where each edge $e \in E$ has a *weight* $w(e) \geq 0$.

For any graph G that is not a tree, there are some edges such that G will remain connected even if we delete these edges. By deleting suitable edges while maintaining the connectivity of the roadmap, and not allowing any distance to grow by more than a factor of t, we obtain a subgraph satisfying the requirements of a spanner.

A few decades ago, Cohen [8] developed an algorithm to construct a spanner of an *unweighted* graph using tree structures. The basic idea is to use a certain probability distribution to select a subset of vertices as roots of trees with depth at most $(k-1)$; then cross-connect the trees to ensure that any two vertices that were adjacent in the input graph are at distance at most $(2k-1)$. The resulting subgraph is a $(2k-1)$-spanner.

To extend this algorithm to weighted graphs, we first normalize the edge weights so that the least and greatest weights are 1 and $\hat{w}$, respectively. We fix a constant $\varepsilon > 0$ and partition the edges of the weighted graph G into $\ell = \lceil \log_{1+\varepsilon} \hat{w} \rceil$ parts, creating edge-disjoint subgraphs $G_1, \ldots, G_\ell$: an edge e with weight $w(e)$ is put into G_i if $(1+\varepsilon)^{i-1} \leq w(e) < (1+\varepsilon)^i$. For each subgraph G_i, we construct a t-spanner $\hat{G}_i$ as if G_i were unweighted; the union $\bigcup_{i=1}^{\ell} \hat{G}_i$ is easily shown to be a $(1+\varepsilon)t$-spanner of the original weighted graph.

Cohen's algorithm influenced subsequent work on spanner algorithms using tree structures [23]; including recent streaming spanner algorithms developed in the graph theory community [10], [11]. Most notably, an algorithm developed by Elkin [10] has constant processing time $O(1)$ per edge on unweighted graphs, and can be extended to weighted graphs using the above partitioning strategy. In this work we apply and improve Elkin's algorithm to construct sparse roadmaps.

A. Streaming spanner algorithm

True streaming graph algorithms process each edge only once, and decide to store or discard the edge immediately. We now describe Elkin's state-of-the-art streaming spanner algorithm for unweighted graphs [10], first at a conceptual level, and then with the necessary details that achieve fast processing and low memory usage.

The algorithm implicitly arranges the n vertices of the input graph into clusters, initially all singletons. As each edge is read, the clustering is modified. Associated with each cluster P is a *base vertex* z_P, which itself may not always belong to the cluster, and an integer *radius* $r(z_P)$ chosen at random from a truncated geometric distribution with the guarantee that $r(z_P) \leq k-1$. An important invariant is that all vertices that have ever been assigned to cluster P are within distance $r(z_P)$ of its base vertex. The algorithm distinguishes between "boundary" vertices, those at distance exactly $r(z_P)$ from z_P, and "interior" vertices, which are closer to the base vertex.

Upon reading an edge (u,v), where u is farther from its cluster's base vertex than v is (ties being broken by vertex ID, say), the algorithm branches as follows. If u is an interior vertex, then v is reassigned to u's cluster and the edge is retained. If u is a boundary vertex, then the edge is retained—and called a *cross edge*—if it is the first such edge between v and u's cluster; otherwise the edge is discarded.

Algorithm 1: ReadEdge$((u,v))$ [10]

```
Let $u$ be the vertex such that $P(u) \succ P(v)$;
if $P(u)$ is a selected label then
    $P(v) \leftarrow P(u) + n$;
    return true;
else if $B(P(u)) \notin M(v)$ then
    $M(v) \leftarrow M(v) \cup \{B(P(u))\}$;
    return true;
return false;
```

The end result is clearly a subgraph $\hat{G}$ of the input graph G. Whenever an edge (u,v) is discarded, with u being the "farther" vertex, the algorithm will have retained a cross edge (x,v) for some vertex x that at some point belonged to u's cluster, P. Therefore, $\hat{G}$ will have a path from u to z_P to x to v, with length at most $r(z_P) + r(z_P) + 1 \leq 2k-1$, by the invariant. This ensures that $\hat{G}$ is a $(2k-1)$-spanner for G.

To implement this algorithm one needs to maintain the clustering information, as well as distance information from base vertices, efficiently. For this, Elkin uses an integer *label* $P(v)$ at each vertex v that encodes both pieces of information as follows. For a label P, its *base value* $B(P) = P \bmod n$ gives the cluster number, while its *level* $L(P) = \lfloor (P-1)/n \rfloor$ gives the distance from vertices with that label to the base vertex of their current cluster. Each vertex v gets a unique initial label $I(v)$ from the set $\{1, 2, \ldots, n\}$. The labeled vertices are totally ordered as follows: we let $P(u) \succ P(v)$ if either $P(u) > P(v)$ or else $P(u) = P(v)$ and $I(u) > I(v)$. Additionally, the algorithm needs to keep track of whether a vertex has previously been connected to a particular cluster. This is done by keeping a list $M(v)$, initially empty, of base values of labels (identifying the corresponding clusters).

The random radius at each vertex is chosen as follows. Put $p = ((\log n)/n)^{1/k}$. For each vertex v, independently, we choose $r(v)$ according to the distribution given by $\mathbb{P}(r(v) = i) = p^i(1-p)$, for $0 \leq i \leq k-2$, and $\mathbb{P}(r(v) = k-1) = p^{k-1}$. The base vertex z_P corresponding to label P is the unique vertex such that $I(z_P) = B(P)$. A label P is said to be *selected* if $L(P) < r(z_P)$; vertices with selected labels are precisely the aforementioned "interior" vertices.

The appropriate manipulation of labels that implements the previously described idea is shown in Algorithm 1. Returning true/false indicates that an edge is to be retained/discarded respectively. This pseudo-code is close to that given by Elkin, except that his version maintains more information explicitly, for notational purposes in the analysis. We refer the reader to his paper [10] for this analysis, which shows that the size of the spanner is $O(kn^{1+1/k}(\log n)^{1-1/k})$ with high probability.

Figure 1 shows an example run of this algorithm on a small instance: a complete graph on 9 vertices, with the setting $k = 3$, so that we are computing a 5-spanner. In this example, the randomly chosen radii include $r(D) = 2$, $r(F) = r(I) = 1$. Notice that each radius is at most $k-1$. The edges are then read in an order that causes the final assignment of labels to (implicitly) define three clusters as shown, with D, F, and I being their base vertices.

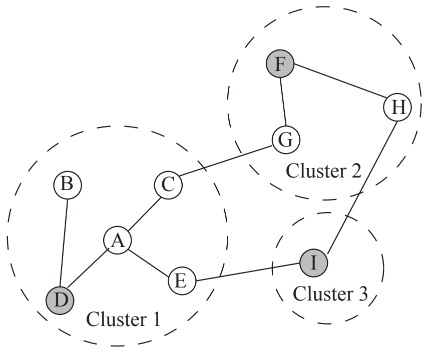

Fig. 1: An example of a 5-spanner (corresponding to $k = 3$) of a complete connected graph. The final clustering, determined by base values of vertex labels, is shown with base vertices shaded. Clusters 1, 2, 3 have radii 2, 1, 1, respectively, so that C, E, G, H are boundary vertices while the rest are interior vertices. Edges (C,G), (H,I), and (E,I) are cross edges.

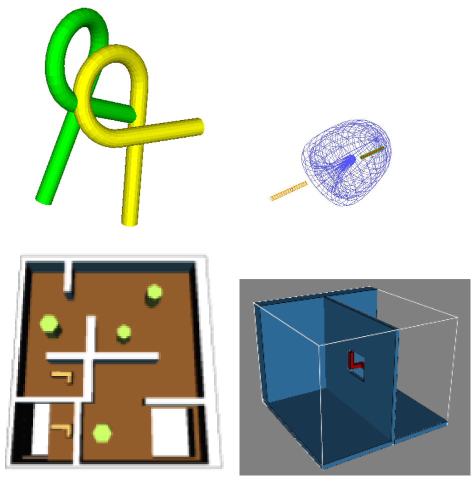

Fig. 2: Four environments. From top left to lower right: Alpha, bugtrap, cubicles, Easy.

III. Incrementally constructing spanners with PRM* and ReadEdge

The algorithm developed by Elkin is one of the fastest spanner algorithms, with a runtime that is only constant per edge. We first applied this streaming spanner algorithm for weighted graphs to construct sparse spanner roadmaps incrementally. In this section, we show how to apply this algorithm with PRM* [15], and also discuss experimental results, which show that the number of edges stored was unfortunately large when used with the weighted, already-somewhat-sparse graphs generated by PRM*.

A. Combining PRM* and ReadEdge

To initialize the combined PRM spanner algorithm, we first need to know n, the number of samples (vertices of the graph). This number will be used to assign radii and initial labels for each vertex. We also need to find (or somehow enforce) the minimum and maximum edge lengths in the graph constructed by the PRM; these distances will be used to normalize the edge lengths, compute the number of required subgraphs l, and assign edges to various subgraphs as each edge is processed. If we use PRM*, we can easily estimate the maximum weight of an edge based on the PRM*'s radius bound.

The minimum edge weight in a roadmap can be found by sampling the vertices before running the PRM, and checking all distances between pairs of vertices, without collision detection. Alternatively, a quasi-random sampling strategy could be used [6], allowing the minimum weight to be computed directly.

During execution of the PRM, after computing a potential edge, the spanner algorithm can be used to determine whether or not to keep the edge; if the edge is stored, some bookkeeping is done to update the labels for vertices involved.

The slowest step in roadmap construction, collision detection, is called when two vertices are being connected. In PRM, the local planner computes a distance between two vertices, and also performs the collision detection. However, if we separate these two steps, then we can avoid the collision detection step if the spanner algorithm decides to discard the edge based on the distance returned by the local planner.

B. Selection of roadmap size

The sharp reader may observe that since Elkin's algorithm requires that n be known in advance, the combination of PRM* and ReadEdge is not truly incremental. This has some disadvantages:

1) We may not know how to choose the number of vertices necessary for a particular environment.
2) It is meaningless to consider the asymptotic optimality of the PRM* metric, if the number of vertices is a fixed constant.
3) The number of levels of subgraphs is determined by the minimum distance between vertices; for large n, the number of subgraphs (though logarithmic) could become large, if samples are chosen unluckily.

All of these difficulties can be overcome quite easily. Choose an initial n. Use a quasi-random sequence to generate the samples; for the current value of n, this allows the minimum distance between vertices to be computed directly. Construct a roadmap for the current value of n, and store this roadmap. If there is time remaining, double n and repeat the process. Since the runtime of each iteration for a fixed n is super-linear, the total run-time is less than twice that of the run-time for the largest roadmap; the asymptotic runtime is not worse than if we knew the 'correct' value for n in advance.

C. Experiments with PRM* filtered by ReadEdge

We used the Open Motion Planning Library (OMPL) [26] to conduct experiments, with a few different 3D environments: Alpha, Bugtrap, Cubicles, Easy.

For each environment, we sampled 50000 vertices, and connected them using PRM*. Table I shows the results of filtering with different values for k (recall that the stretch is $2k-1$), including how many edges (relative to the original PRM*) were stored. We used $\varepsilon = 0.1$ (the approximation

	Alpha	Bugtrap	Cubicles	Easy
k = 2	96.5%	96.2%	90.2%	96.0%
k = 3	86.8%	89.7%	87.3%	81.9%
k = 4	79.9%	83.4%	78.9%	75.2%
k = 5	69.6%	81.0%	72.3%	69.0%
k = 6	67.3%	77.9%	70.4%	64.6%

TABLE I: Streaming $(2k-1)$-spanners on different environments. The table shows the percent of edges stored for different k. The original dense roadmaps for different environments contain between 4 million and 26 million edges.

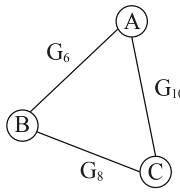

Fig. 3: An example of redundant edges being stored in the original spanner algorithm for weighted graphs.

term that appears for weighted graphs) for all the experiments. The results are disappointing: the spanners generated stored many more edges than expected—between 60% and 90% as many as the original graph, depending on the selected stretch.

IV. Improved spanners with PRM* and PropagatedReadEdge

Careful examination of the spanner roadmaps reveals some apparent redundancy in the edges kept, due to the separation of the weighted graph into several subgraphs based on edge weights, using the basic technique described in Section II. We now propose an algorithm that addresses this redundancy, while remaining provably correct in terms of its stretch guarantee.

To understand the issue, consider an edge $e=(u,v)$ that belongs to G_j, such that when this edge is read we have already retained a path in G_i (for some $i<j$) between u and v, but there is no such path in G_j. According to Elkin's algorithm, we would definitely retain e in the spanner for G_j. However, each edge in G_i has smaller weight than each edge in G_j. Therefore, if the u–v path in G_i has length at most $2k-1$, then in the weighted graph G its length is at most $(2k-1)w(e)$, which means that we can safely discard edge e without violating the stretch requirement.

For a concrete example, consider three vertices as in Figure 3: edge $(A,B)\in G_6$, $(B,C)\in G_8$, and $(A,C)\in G_{10}$. Procedure ReadEdge from Algorithm 1 (as applied to weighted graphs) stores all three edges, since no subgraph knows the connection information in any other subgraph. However, if (A,B) and (B,C) arrive earlier than (A,C), then for a large enough stretch parameter (A,C) need not be stored since A and C are already connected and at distance below the stretch.

To achieve a reduction in redundancy, we revisit the way that ReadEdge modifies the clustering of vertices when it reads an edge. In our improved algorithm, we allow modifications in the clustering at one subgraph G_i to *propagate* to all "higher" subgraphs G_j, with $j>i$. To be precise, we maintain a label $P_i(v)$ and a cross-connection list $M_i(v)$ for each index i, corresponding to subgraph G_i. Intially, $P_i(v)=I(v)$ and $M_i(v)=\varnothing$ for each i. These are then manipulated as shown in Algorithm 2. Recall that the edge weights are normalized to lie in the interval $[1,\hat{w}]$, and notice that the formula $1+\lfloor\log_{1+\varepsilon}w\rfloor$ identifies the index q of the subgraph G_q to which an edge with weight w belongs.

Algorithm 2: PropagatedReadEdge

```
Input: Edge $(u,v)$ with weight $w$
$q \leftarrow 1+\lfloor\log_{1+\varepsilon}w\rfloor$ ;
Let $u$ be the vertex such that $P_q(u) \succ P_q(v)$;
if $P_q(u)$ is a selected label then
    /* Update labels: */
    for $i=q$ to $l$ do
        Select $u$ as the vertex such that $P_i(u) \succ P_i(v)$;
        if $L(P_i(u)) < (t-1)$ then
            $P_i(v) \leftarrow P_i(u)+n$;
    return true;
else if $B(P_q(u)) \notin M_q(v)$ then
    /* Update cross-connection list: */
    $M_q(v) \leftarrow M_q(v) \cup \{B(P_q(u))\}$;
    return true;
return false;
```

When applied to the roadmap construction, once we decide to store the edge, we do not update the labels or cross-connection lists at once. We return **true**, and run collision detection. If the collision detection indicates there is no collision along the path, we then update the corresponding label or cross-connection list. Additional processing can be applied to algorithm 2, such as broadcasting cross-connection list (increasing storage space), but only a limited number of edges are additionally discarded. In algorithm 2, we choose the most simple and effective modification.

Algorithm 2 should be able to avoid saving some redundant edges, but it does not increase storage space. (For details of the storage space and the expected number of edges, please reference [10].) The algorithm propagates selected information between different subgraphs. When an edge (u,v) is stored, it might belong to a tree (update the labels), or cross connection (append to a cross connection list). Updating labels cannot increase the space requirements, since for every vertex a label is maintained in each subgraph in the original algorithm. We do not propagate information about cross connections, so this cannot cause a blow-up in storage either.

We now prove that our algorithm correctly computes a spanner. For this we need a more sophisticated version of the invariant described in Section II. Recall that for a label P, z_P is the unique vertex such that $I(z_P)=B(P)$ and that $r(v)$ is the randomly assigned radius for vertex v.

Lemma 1: Let v be a vertex of G and let j be an index of some subgraph G_j. If $P_j(v)$ ever takes the value P, then in the unweighted graph $H_j=G_1\cup\cdots\cup G_j$, the distance between v and z_P is at most $L(P)$. Consequently, this distance is at most $r(z_P)$.

Proof: The latter conclusion follows from the former, because by design (and by definition of "selected label") we have $L(P) \leq r(z_P)$ for any label P that is ever used. For the former conclusion, we use induction on $L(P)$. When $L(P) = 0$, P must be the initial label $I(v)$. Thus $z_P = v$ and indeed v is at distance 0 from z_P.

Suppose $L(P) > 0$. Then $P_j(v)$ was assigned the value P upon reading an edge (u,v) with $P_j(u) = P - n$. Since $L(P-n) = L(P) - 1$ and $B(P-n) = B(P)$, by the inductive hypothesis, u is at distance at most $L(P) - 1$ from z_P in the graph H_j. By design of PropagatedReadEdge, the edge (u,v) falls in some subgraph G_q with $q \leq j$, so this edge is in H_j as well. Thus, the distance between v and z_P is at most $L(P)$, completing the proof. ■

Theorem 2: Algorithm 2 applied to a weighted graph generates a $(1+\varepsilon)(2k-1)$-spanner.

Proof: Let $\hat{G}$ be the weighted graph created by the algorithm. It suffices to prove that for an arbitrary edge $e = (u,v)$ that is not retained, there is a path in $\hat{G}$ of weighted length at most $(1+\varepsilon)(2k-1)w(e)$. Suppose edge e belongs to G_q and $P = P_q(u) \succ P_q(v)$. We must have $B(P) \in M_q(v)$, so we must have retained some cross edge (v,x) belonging to G_q, where x had a label satisfying $B(P_q(x)) = B(P)$. By the invariant in Lemma 1, both u and x are at distance at most $r(z_P)$ from z_P. Therefore, following a path from u to z_P to x to v, we see that v is at distance at most $2r(z_P) + 1 \leq 2k - 1$ from u, in the retained subgraph of the unweighted graph $H_q = G_1 \cup \cdots \cup G_q$.

Every edge in H_q has weight at most $(1+\varepsilon)^q$, whereas $w(e) \geq (1+\varepsilon)^{q-1}$. It follows that the weighted distance from u to v in $\hat{G}$ is at most $(1+\varepsilon)(2k-1)w(e)$, as required. ■

A. Processing time per edge

As we already know, the original streaming spanner algorithm on unweighted graphs has constant processing time per edge. However, when we extend the algorithm to weighted graphs, the processing time per edge is no longer $O(1)$, and becomes worst case $O(\ell)$ (where ℓ is the number of subgraphs as defined in Section II, $\ell = \lceil \log_{1+\varepsilon} \hat{w} \rceil$). The number of subgraphs is typically small; in our experiments, we found that for graphs with as many as 50,000 vertices, ℓ is less than 50.

Analysis of the amortized run-time suggests that the processing time per-edge is on average much less than $O(\ell)$. If (u,v) belongs to a tree, a for loop will be invoked which will run in $O(\ell)$. For a graph with n vertices, there are at most $O(n)$ edges that could belongs to some tree. For a graph with $|E|$ edges, to process all edges, we need $O(|E| + n \cdot \ell)$ operations. Then, the amortized runtime per edge is $O(1 + n \cdot \ell / |E|)$. For a PRM* roadmap, $|E| = O(n \log n)$, so the amortized runtime per edge is $O(\max\{\ell / \log n, 1\})$.

In practice, we find that the runtime of improved algorithm is very close to the original algorithm, which has constant processing time per edge. To process a roadmap with 50000 vertices, ReadEdge takes about 23 seconds, and PropagatedReadEdge takes about 26 seconds.

B. Experimental results for improved spanner algorithm

We applied SS-PRM* on six environments using OMPL, including the four environments previously tested with ReadEdge. The same PRM* parameters were adopted, and similar roadmaps were generated. The number of edges on the roadmaps ranged from 4 million to 26 million. The ratio of edges stored by SS-PRM*, for various stretch values, is shown in table II. Compared to the results for ReadEdge, these roadmaps are much sparser.

	k=2	k=3	k=4	k=5	k=6
Alpha	20.7%	7.68%	4.99%	4.76%	3.96%
Apartment	39.1%	14.1%	8.71%	7.82%	6.56%
Bugtrap	20.9%	4.21%	3.17%	2.10%	2.00%
Cubicles	48.6%	42.8%	21.4%	14.6%	12.8%
Home	49.1%	42.0%	26.5%	16.9%	14.8%
Easy	56.3%	40.4%	26.1%	20.4%	15.1%

TABLE II: Results of running SS-PRM* on different environments with different k to construct $(2k-1)$-spanners. Each roadmap contains 50000 vertices.

	PRM*	2 SS-PRM*	4 SS-PRM*	6 SS-PRM*	PRM
Alpha	160	48	40	36	23
Apartment	152	70	60	55	25
Bugtrap	60	32	26	22	20
Cubicles	170	94	86	80	22
Home	117	93	77	70	26
Easy	101	61	54	49	24

TABLE III: Runtime comparison (in minutes) between PRM*, SS-PRM* for different k, and PRM that connects to 15 nearest neighbors.

	k=2	k=3	k=4	k=5	k=6
Alpha	104%	135%	136%	142%	154%
Apartment	103%	104%	105%	107%	110%
Bugtrap	102%	107%	109%	115%	118%
Cubicles	101%	104%	106%	108%	110%
Home	101%	104%	105%	107%	110%
Easy	101%	103%	106%	109%	111%

TABLE IV: Average increase of path lengths. The table shows the average route length for SS-PRM* over the length of corresponding route for PRM*. The experiments are conducted on corresponding environments with 5000 vertices instead of 50000 vertices. In each resulting roadmap, we calculated all-pair-shortest paths to compare the lengths of different paths.

C. Runtime of PRM, PRM, and SS-PRM**

We ran 15-nearest-neighbor PRM, PRM* by itself, and SS-PRM* for different k, and compared the runtimes, as shown in table III. The experiments were run on a modern desktop computer. The runtime of SS-PRM* is much less than that of the full PRM*, but not necessarily proportional to the number of edges processed and stored, since the collision detection only ran on the edges the algorithm considered storing. Even though the runtime of SS-PRM* is not as fast as the original PRM, note that PRM only connects 15 neighbors while PRM* (and SS-PRM*) may connect each vertex to thousands of neighbors.

D. Average stretch

In spite of the few edges stored and the stretch we enforced, we would like to know how much the algorithm relaxed different routes in the average case. PRM* and SS-PRM* were run on the same environments with each roadmap containing 5000 vertices. All-pairs shortest paths were calculated on each roadmap (because of the high time cost of all-pairs shortest paths, experiments on roadmaps with

	k=2	k=3	k=4	k=5	k=6
Alpha	2.23	1.54	1.32	1.25	1.19
Apartment	1.42	1.33	1.28	1.13	1.06
Bugtrap	0.89	0.76	0.63	0.58	0.55
Cubicles	2.73	2.52	2.30	1.94	1.87
Home	2.26	2.03	1.97	1.95	1.89
Easy	2.01	1.97	1.87	1.66	1.54

TABLE V: The median of the route length of 15-neighbor PRM over the route length of SS-PRM* with different k.

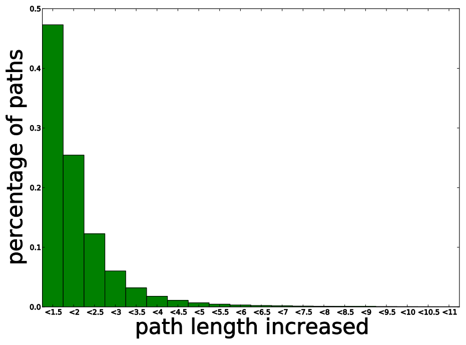

Fig. 4: For a roadmap with 5000 vertices on Alpha environment with $k = 6$, the paths in spanner increased with different percentage. The figure showed how many edges belong to each different range. The average ration (path ratio) is 2.0. In the original roadmap, there are 0.4 million edges, and there are about 0.06 million edges left in the spare roadmap.

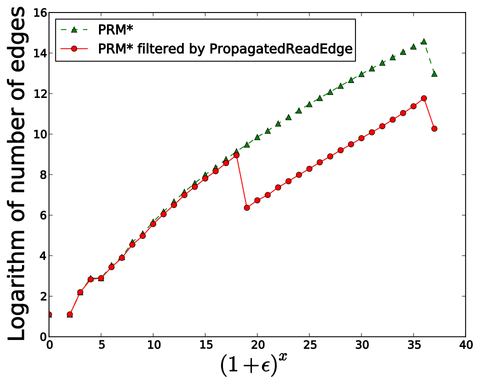

Fig. 5: The weight distribution of edges in the roadmaps returned by SS-PRM*.

Environment	No. of vertices	IRS	SS-PRM*
Bugtrap	5000	18.5	0.9
	10000	98.8	5.1
	20000	325.0	15.3
Apartment	5000	35.2	5.5
	10000	129.2	14.8
	20000	499.1	29.9

TABLE VI: Runtime comparison in minutes between IRS and SS-PRM* on two environments with 5000, 10000, and 20000 vertices.

50000 vertices are infeasible) to compare the average route length increase. Experiments show that even when the stretch is large (when $k = 6$, the stretch is 11), on average the route lengths increased very little. Figure 4 shows the distribution of route cost increases on a roadmap generated for the Alpha environment with 5000 vertices.

We hypothesize the reason the average route length increased by very little compared to the worst bound is as follows: short edges are mostly stored while many long edges are discarded. Only the routes that are originally short are approximated by the factor close to the worst bound (stretch), while the routes that are long are approximated by many short edges, with length similar to the original cost. The weight distribution, e.g. the number of edges belonging to each subgraph, is plotted in figure 5, and lends some support to this hypothesis. We also see a sudden drop of number of edges stored in 19th subgraph and above. We believe this to be because lower-level subgraphs have fewer edges and are already sparse.

We also compared the route lengths between roadmaps returned by 15-nearest-neighbor PRM, and SS-PRM*. The median route length of 15-nearest-neighbor PRM over the median route length of SS-PRM* for different k is shown in table V.

E. Comparison to the Incremental Roadmap Spanner

We compared the results to results reported for the Incremental Roadmap Spanner (IRS) algorithm in [14], [19]. For small k, such as $k = 2$, IRS stored $10 - 25\%$ of the original edges, while SS-PRM* stored about $30 - 60\%$; however, for $k = 6$, IRS algorithm stored about $4 - 10\%$ of the original edges, and SS-PRM* stored $2 - 15\%$ of original edges. Overall, SS-PRM* stored more edges when k is small, but the percentage drops rapidly as k grows.

On the other hand, the run time of SS-PRM* is less than that of IRS [19]. For each edge, PropagatedReadEdge has $O(\max\{\ell/\log n, 1\})$ processing time, and each vertex has $O(\log n)$ neighbors, so for each of the n vertices, our algorithm requires $O(\max\{\ell, 1\}) = O(\max\{\log_{1+\varepsilon} \hat{w}, \log n\})$, while IRS [19] requires $O(n \cdot \log^2 n \cdot \log\log n)$ per vertex in the worst case.

We used IRS code provided by the authors of [19], and ran some experiments to compare runtime between IRS and SS-PRM*. Results are shown in table VI. Due to the lengthy run-time of IRS, we were not able to conduct experiments with more than 20,000 vertices.

Significantly, we can see that for some number of vertices, the runtime of IRS will dominate the run-time, while for SS-PRM*, collision detection remains the dominant term.

F. Graph size and density affect PropagatedReadEdge

The performance (the number of edges stored) of the streaming spanner algorithm is greatly affected by both size and the density of the original graph. To explore this, we randomly generated different (unweighted) roadmaps with different sizes (number of vertices, figure 6) and density (number of neighbors, figure 7).

From figures 6 and 7, it is not hard to conclude that for larger and denser graphs, the streaming spanner algorithm performs better, as will PropagatedReadEdge. Therefore, for even larger and denser roadmaps (more complex environments and more complex robots), we expect SS-PRM* to still

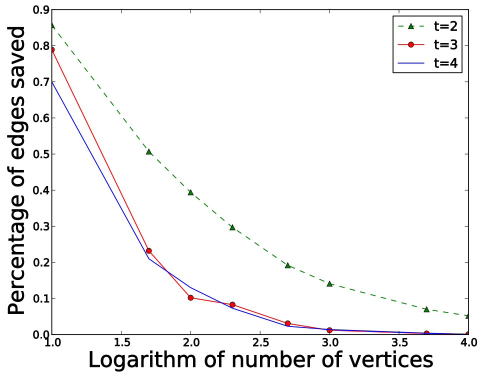

Fig. 6: For random roadmaps with different number of vertices, the percentage of edges stored for different k in a $(2k-1)$-spanner using streaming spanner algorithm.

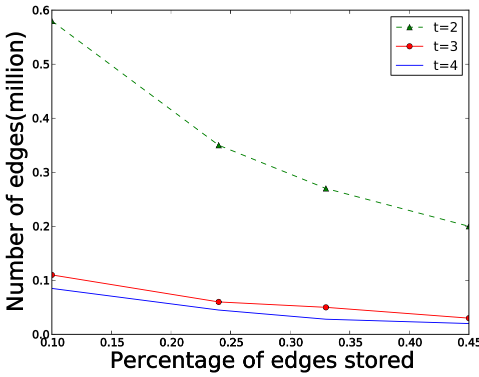

Fig. 7: For random roadmaps with 1000 vertices, but different number of edges (density), the percentage of edges stored using streaming spanner algorithm with different k.

produce roadmaps computationally efficiently while storing an even smaller percentage of edges.

V. Conclusions and Future Work

In this paper, we introduced a state-of-art streaming spanner algorithm for pruning graphs representing distances and connectivity. The algorithm stores a number of edges that is competitive with the IRS algorithm, but has a processing time that is essentially constant per vertex.

Previous work on sparse roadmaps has been applied to symmetric systems, with undirected graphs; we would like to extend our work to directed graphs. Current directed spanner algorithms are off-line rather than streaming, however. We have developed an algorithm that can construct directed spanners incrementally, but the size of the resulting spanner is currently theoretically unbounded, even though experimental results are promising.

VI. Acknowledgments

This work was funded by a grant from the Neukom Institute for Computational Science. We would also like to thank Kostas Bekris and Yu-Han Lyu for helpful discussions and feedback.

References

[1] Surender Baswana. Streaming algorithm for graph spanners - single pass and constant processing time per edge. *Inf. Process. Lett.*, 106(3):110–114, 2008.

[2] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. Fully dynamic randomized algorithms for graph spanners. *ACM Transactions on Algorithms*, 8(4):35, 2012.

[3] Arnab Bhattacharyya and Konstantin Makarychev. Improved approximation for the directed spanner problem. *CoRR*, abs/1012.4062, 2010.

[4] Prosenjit Bose, Vida Dujmovic, Pat Morin, and Michiel H. M. Smid. Robust geometric spanners. *CoRR*, abs/1204.4679, 2012.

[5] Michael S. Branicky, Steven M. Lavalle, Kari Olson, and Libo Yang. Quasi-randomized path planning. In *In Proc. IEEE Intl Conf. on Robotics and Automation*, pages 1481–1487, 2001.

[6] Michael S. Branicky, Steven M. LaValle, Kari Olson, and Libo Yang. Quasi-randomized path planning. In *ICRA*, pages 1481–1487. IEEE, 2001.

[7] J. F. Canny. *The Complexity of Robot Motion Planning*. MIT Press, Cambridge, MA, 1988.

[8] E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. In *Proceedings of the 1993 IEEE 34th Annual Foundations of Computer Science*, FOCS '93, pages 648–658, Washington, DC, USA, 1993. IEEE Computer Society.

[9] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000.

[10] Michael Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Trans. Algorithms*, 7(2):20:1–20:17, March 2011.

[11] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. Graph distances in the streaming model: the value of space. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '05, pages 745–754, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[12] David Hsu, Lydia E. Kavraki, Jean-Claude Latombe, Rajeev Motwani, and Stephen Sorkin. On finding narrow passages with probabilistic roadmap planners. In *Workshop on Algorithmic Foundations Robotics*, pages 141–153, Natick, MA, 1998.

[13] David Hsu, Robert Kindel, Jean claude Latombe, and Stephen Rock. Randomized kinodynamic motion planning with moving obstacles, 2000.

[14] Marble J and Bekris KE. Asymptotically near-optimal planning with probabilistic roadmap spanners. *IEEE Transactions on Robotics*, 29(3), 2013.

[15] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7), 846-894, 2011.

[16] Lydia Kavraki, Petr Svestka, Jean claude Latombe, and Mark Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In *IEEE International Conference on Robotics and Automation*, pages 566–580, 1996.

[17] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, UIUC, 1998.

[18] Tomas Lozano-Perez. Spatial planning: A configuration space approach, 1980.

[19] J. Marble and K. E. Bekris. Asymptotically near-optimal is good enough for motion planning. In *Proc. of the 15th International Symposium on Robotics Research (ISRR-11)*, 28. Aug. - 1 Sep 2011.

[20] James D. Marble and Kostas E. Bekris. Computing spanners of asymptotically optimal probabilistic roadmaps. In *IROS*, pages 4292–4298, 2011.

[21] James D. Marble and Kostas E. Bekris. Towards small asymptotically near-optimal roadmaps. In *ICRA*, pages 2557–2562, 2012.

[22] David Peleg and Alejandro A. Schffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

[23] Iam Roditty, Mikkel Thorup, and Uri Zwick. Roundtrip spanners and roundtrip routing in directed graphs. *ACM Trans. Algorithms*, 4(3):29:1–29:17, July 2008.

[24] Liam Roditty. Fully dynamic geometric spanners. *Algorithmica*, 62(3-4):1073–1087, 2012.

[25] J. T. Schwartz and M. Sharir. On the Piano Movers' Problem: I. The case of a two-dimensional rigid polygonal body moving amidst polygonal barriers. *Communications on Pure and Applied Mathematics*, 36:345–398, 1983.

[26] Ioan A. Sucan, Mark Moll, and Lydia E. Kavraki. The open motion planning library. *IEEE Robotics and Automation Magazine*, 19(4):72082, December 2012.