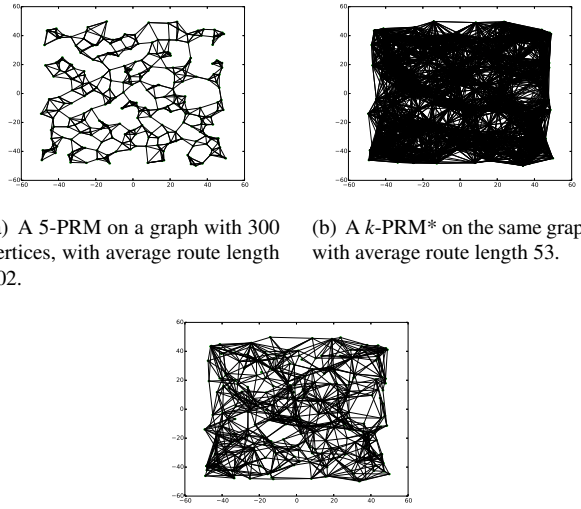
A fast online spanner for roadmap construction

Weifu Wang Devin Balkcom Amit Chakrabarti

Abstract

This paper introduces a fast *weighted streaming spanner* algorithm (WSS) that trims edges from roadmaps generated by robot motion planning algorithms such as Probabilistic Roadmap (PRM) and variants (*e.g.* k-PRM*) as the edges are generated, but before collision detection; no route in the resulting graph is more than a constant factor larger than it would have been in the original roadmap. Experiments applying WSS to k-PRM* were conducted, and the results shows our algorithm's capability to filter graphs with up to 1.28 million vertices, discarding about three-quarters of the edges. Due to the fact that many collision detection steps can be avoided, the combination of WSS and k-PRM* is faster than k-PRM* alone.

The paper further presents an online directed spanner algorithm that can be used for systems with non-holonomic constraints, with proof of correctness and experimental results.



(a) A 5-PRM on a graph with 300 vertices, with average route length 102.

(b) A k-PRM* on the same graph, with average route length 53.

(c) Application of a 9-spanner developed in this paper to k-PRM*. Average route length is 69.

Figure 1: Application of the online spanner algorithm (WSS) to a roadmap generated by k-PRM*.

1 Introduction

Probabilistic roadmap (PRM) algorithms (Kavraki et al. [1996, 1998]) have been applied successfully in practice to fairly high-dimensional motion planning problems for almost two decades. In the learning phase, PRM algorithms place samples in the configuration space, and attempt to connect each new sample to existing samples using a *local planner*, generating a *roadmap* graph. In the query phase, a given start and goal are connected to the roadmap, and the graph is searched for a connecting path.

Probabilistic roadmaps typically contain edges that could be removed without affecting the connectivity of the graph, or the cost of paths in the graph "too much". This paper presents *streaming spanner algorithms* that avoid adding some of these edges to the graph. We expect smaller graph representations of configuration spaces to be easier to compute, store, transmit over a network, and search for a path.

One common approach to limiting memory and computational requirements of PRM is to limit the number of connections attempted to some fixed number k of (approximately) nearest neighbors: a k-PRM (Kavraki et al. [1996]). Choosing some fixed value k can be problematic; analysis in Kavraki et al. [1998] is limited to the case where each new sample connects to all previous samples.

The recent PRM* and k-PRM* algorithms (Karaman and Frazzoli [2011]) give a more principled way to make an appropriate number of connections. PRM* attempts to make all connections to samples in a ball of radius r, and chooses the radius of the ball dynamically in such a way that path-lengths in the roadmap asymptotically approach optimality. This is a very strong result, essentially giving in some sense the minimum number of connections required to achieve asymptotic optimality. k-PRM* directly chooses a number of neighbors (rather than using a radius) that increases with the number of vertices.

In 2011, Marble and Bekris [2011a] introduced the concept of *spanners* into the robotics community to reduce the density of PRM* roadmaps (at some cost in path optimality), and have improved their approach over the past two years (Marble and Bekris [2011b, 2012], Krontiis et al. [2012], Marble and Bekris [2013]). A spanner of a graph is a subgraph that contains all vertices, but fewer edges, such that the distance between any two points in the spanner is no more than a constant t (the *stretch*) times the distance in the graph.

This paper, which extends work first presented in Wang

et al. [2013], introduces an online spanner algorithm based on the streaming spanner of Elkin [2011]. We extend Elkin's algorithm to better handle weighted "sparse" graphs; we call the modification the Weighted Streaming Spanner (WSS).

The primary advantage of WSS over previous spanner algorithms that have been applied to sampling-based planners, such as the Incremental Roadmap Spanner (IRS) (Marble and Bekris [2012]), is speed. The algorithms we present have an amortized processing time that is constant per edge. This means for problems with $50,000$ samples, the algorithm we present, WSS, requires time on the order of seconds, while IRS may require time on the order of hours. In fact, we have been able to apply WSS to a problem with 1.28 million samples, on a standard laptop. It must be noted that in our experiments, we found that for graphs of a size where IRS can be practically applied, IRS produces spanners with somewhat fewer edges.

The paper has three parts. First, we briefly present Elkin's algorithm and show some experimental results. Second, we present an improved algorithm, prove its correctness, and explore its application experimentally. Finally, we prove some results showing limitations of directed spanners, propose a directed spanner algorithm, and show experimental results for Dubins car (Dubins [1957], a system with non-holonomic constraints) among obstacles. To our knowledge, this is the first directed spanner to be applied to generate more compact roadmaps for systems with non-holonomic constraints.

1.1 Related Work

Probabilistic roadmaps (PRM) were introduced in the mid-1990s (Kavraki et al. [1996]) as a sampling based motion planning algorithm. Several other sampling based algorithms, such as Rapidly-Exploring Random Trees (RRT) (Lavalle [1998]), and the extension of PRMs to systems with non-holonomic constraints (Hsu et al. [1998, 2002], Svestka and Overmars [1997]) were developed shortly thereafter. Prior to these sampling based algorithms, most motion planning algorithms were deterministic and based on the specific geometric properties of the robots and the configuration spaces (Canny [1988], de Berg et al. [2000], Lozano-Pérez [1983], Schwartz and Sharir [1983]). Compared to these deterministic algorithms, sampling based algorithms are easy to implement and can solve certain high dimensional problems effectively. Different sampling strategies and connection strategies can further improve the performance of a PRM; for example, see Amato et al. [1998], Lien et al. [2003].

Recently, Karaman and Frazzoli [2011] developed another variation, PRM*. The connection strategy of PRM* is based on the number of samples and the volume of the configuration space. The construction time of a PRM* roadmap is typically much higher than that of a PRM with a small fixed number of edges per sample, and PRM* roadmaps may be too dense to store for many scenarios.

Spanners were first introduced in the late 1980s (Peleg and Schffer [1989]). For any given graph G, a t-spanner for G is a subgraph of G that approximates the distance metric of G using fewer edges. In 1993, Cohen developed a spanner algorithm based on tree structures (Cohen [1998]); the trees efficiently represent local connectivity of the space. Following Cohen's algorithm, several offline spanner algorithms were developed based on similar tree-like structures (Roditty et al. [2008], Thorup and Zwick [2001]).

Recently, streaming spanner algorithms were developed such that on the arrival of an edge, the algorithm can decide immediately whether to store the edge. In 2005, Feigenbaum et al. [2005] introduced a streaming spanner algorithm. Elkin [2011] and Baswana (Baswana [2008], Baswana et al. [2012]) then developed faster algorithms that process each edge in $O(1)$ and amortized $O(1)$ time, respectively. The previously mentioned algorithms were designed for unweighted graphs, but can be extended to weighted graphs and geometric graphs. Geometric spanners can also directly be found using other algorithms (Roditty [2012], Bose et al. [2012]).

The core algorithm we present is strongly based on an algorithm by Elkin (Elkin [2011]). Elkin's algorithm is fast, but PRM* roadmaps are *already* sparser than Elkin's analysis predicts computed spanners to be, and experimentally, we found that Elkin's algorithm reduced the number of edges stored only by $10-35\%$ on the cases we tried.

For directed graphs, spanners are harder to find. Algorithms we are aware of are offline, and many are based on Integer Linear Programming (see Bhattacharyya and Makarychev [2010]). Other algorithms can only find spanners based on a modified metric (Roditty et al. [2008]).

Spanner algorithms discard edges while keeping all the vertices. Even though discarding vertices is not the focus of this paper, there do exist algorithms in the robotics community that construct sparse roadmaps by using fewer vertices, either using connectivity ("visibility") to prevent the vertices from being added to the roadmaps (Laumond and Nissoux [2000]), or discarding vertices after the dense roadmaps are constructed (Marble and Bekris [2012], Krontiis et al. [2012], Shaharabani et al. [2013]).

2 Spanners

For any graph that is not a tree, some edges can be deleted without affecting the connectivity. The distances between some pairs of vertices may increase due to the deletion of these edges. A t-spanner is constructed by deleting some carefully selected edges, such that the maximum increase of the distances between any pair of vertices does not exceed a factor of t. Formally,

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

Cohen's algorithm (Cohen [1998]) finds a $(2m-1)$-spanner of an *unweighted* graph using tree structures. The basic idea of the algorithm is to select a subset of vertices V_t and edges, such that all the other vertices can reach one of the vertices in V_t in $m-1$ steps. Now, each vertex belongs to a tree rooted at a vertex in V_t. Then, cross-connect all the trees to ensure the maximum distance between any two originally adjacent vertices is at most $2m-1$. Therefore the resulting graph is a $(2m-1)$-spanner of the original graph. This idea is central to many other spanner algorithms, including that of Elkin [2011].

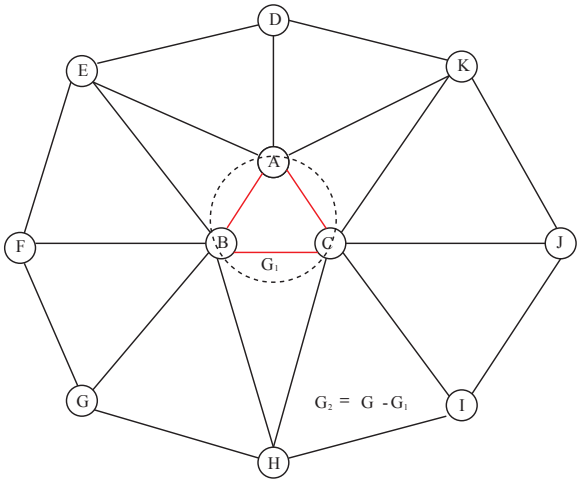


Figure 2: Example of extending spanner algorithms to weighted graphs. In this graph, all edges were partitioned into two subgraphs, G_1 and G_2. All the edges of G_1 are shorter than any of the edges of G_2. The weighted spanner extension finds spanner of G_1 and G_2 while treating them as unweighted. The union of two spanners will be an approximation of the spanner for G.

2.1 Extension to weighted graphs

Any unweighted spanner algorithm can be extended to weighted graphs. First, normalize weights on the graph so that the minimum and maximum weights are 1 and $\hat{w}$ respectively. Then choose a constant $\varepsilon > 0$, and partition the edges into $\ell = \lceil \log_{1+\varepsilon} \hat{w} \rceil$ edge-disjoint subgraphs $G_1, \ldots, G_\ell$. An edge e with weight $w(e)$ belongs to subgraph G_i if $(1+\varepsilon)^{i-1} \leq w(e) < (1+\varepsilon)^i$.

Now construct a $(2m-1)$-spanner $\hat{G}_i$ for each G_i as if G_i were unweighted; the union $\bigcup_{i=1}^{\ell} \hat{G}_i$ is easily shown to be a $(1+\varepsilon)(2m-1)$-spanner of the original weighted graph.

3 Elkin's streaming spanner algorithm

Streaming spanner algorithms process each edge only once, and the decision to retain or discard the edge is made immediately. The streaming model fits roadmap construction perfectly: both k-PRM and PRM* return one edge at a time. What is more, the decision to discard may be made before the collision detection, so finding a spanner of a roadmap may be faster than using k-PRM or PRM* alone.

We first introduce a state-of-the-art streaming spanner algorithm for unweighted graphs developed by Elkin [2011]. We found experimentally that even though Elkin's algorithm is fast, the extension Elkin presents for weighted graphs is not effective in finding sparse spanners for PRM* roadmaps.

3.1 Algorithm description

Algorithm 1 gives a succinct pseudocode description of how edges are processed by Elkin's streaming spanner algorithm. This section describes the definitions required the understand the algorithm, as well as the setup required before the algorithm is run.

To construct a $(2m-1)$-spanner using Elkin's algorithm, all n vertices of the input graph G are implicitly arranged into clusters. All the clusters are initially singletons, and modified as edges are read.

Each cluster C is associated with a *base vertex* denoted as z_C. (Notice, z_C may no longer belong to C at the end of the algorithm even though C will still be described with respect to its base vertex z_C.)

Each cluster has a maximum size. An integer $r(u) \leq m-1$ is assigned to each vertex u using a truncated geometric distribution; this integer (the *radius*) describes how far away a vertex might be and still be included in a cluster centered at u (if u ends up as a cluster base). We will describe the particular distribution below; for now it is sufficient to understand that each vertex has some small integer associated with it.

An important invariant is that, for a cluster C, all the vertices in this cluster are within distance $r(z_C)$ from z_C. Vertices in C that are exactly $r(z_C)$ away from z_C are distinguished as "boundary" vertices, while other vertices in C are referred to as "interior" vertices.

For an incoming edge (u,v), let us assume u is farther from its cluster's base vertex than v (to break a tie, assume u has higher vertex ID); otherwise swap u and v. The algorithm processes this edge as follows. If u is not a boundary vertex, then this edge is retained and v is reassigned to u's cluster. If u is a boundary vertex, the edge is retained if and only if this edge—referred to as a *cross edge*—is the first edge that connects these two clusters through v.

Clearly, the resulting graph $\hat{G}$ is a subgraph of the input graph G. For any edge (u,v) discarded by the algorithm, there must exist an edge (x,v) and a cluster C such that both x and u belong to C at some point (x and u may belong to C at different

Algorithm 1: ReadEdge$((u,v))$(Elkin [2011])

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**;

times). Thus there exists a path from u to x to v that connects u and v, with length no longer than $r(z_C) + r(z_C) + 1 \leq 2m-1$. The condition holds for any pair of vertices u and v, so $\hat{G}$ is a $(2m-1)$-spanner of G.

In order to cluster the vertices and determine which edges to discard, the algorithm needs to keep track of the status of each vertex: which cluster it belongs to, and the distance to this cluster's base vertex. Elkin uses only one integer $P(u)$—called the *label*—at each vertex u to keep track of both pieces of information. The cluster number is given by the *base value* of a label $B(P(u)) = P(u) \bmod n$, and the *level* of a label $L(P(u)) = \lfloor (P-1)/n \rfloor$ gives the distance from u to the base vertex of its current cluster. A unique initial label $I(u)$ is selected from set $\{1,2,\ldots,n\}$ for each vertex. The total order of the labels is given as follows: $P(u) \succ P(v)$ if either $P(u) > P(v)$ or $P(u) = P(v)$ and $I(u) > I(v)$. Each vertex u also keeps an initial empty list $M(u)$ which may contain a collection of base values of labels. A base value $\hat{B} \in M(u)$ indicates u is connected to the cluster whose base vertex has initial label $\hat{B}$.

The radius at each vertex is randomly selected following a truncated geometric distribution, as mentioned earlier. Let $p = ((\log n)/n)^{1/m}$. Then, the distribution is given by $\mathbb{P}(r(u) = i) = p^i(1-p)$, for $0 \leq i \leq m-2$, and $\mathbb{P}(r(u) = m-1) = p^{m-1}$. The base vertex z_C with label P is the unique vertex such that $I(z_C) = B(P)$.

Armed with these definitions, we may consider Algorithm 1. A label P is considered to be *selected* if and only if $L(P) < r(z_C)$; vertices with selected labels are exactly the "interior" vertices mentioned above.

Algorithm 1 shows the implementation of the previously described label manipulation procedure. The returned value—true/false—indicates whether the corresponding edge is retained/discarded respectively. This pseudo-code is mostly the same as that given by Elkin [2011], except that his version maintains more explicit information for the purpose of analysis. The analysis showing that the size of the spanner is $O(mn^{1+1/m}(\log n)^{1-1/m})$ and the space required is $O(m \cdot n^{1+1/m} \cdot (\log n)^{2-1/m})$ with high probability as described in detail in Elkin [2011].

Figure 3 shows several steps of an example run of this algorithm on a small graph: a complete graph on 9 vertices, with the setting $m = 3$ (5-spanner). Captions describe several of the interesting cases when edges are either kept or discarded, and when clusters form or change.

Elkin [2011] extends this algorithm to weighted graphs using the partition method mentioned in section 2.1.

3.2 Elkin's algorithm with k-PRM*

It seems natural to use Elkin's algorithm to prune some edges generated by k-PRM*. We tried this, but the results were unsatisfactory. For example, on the *alpha-puzzle* environment from the OMPL motion planning library (Sucan et al. [December 2012]), we ran a k-PRM* with 40,000 vertices. Over 10 runs, the k-PRM* retained about 850,000 edges on average (Section 5). On average, Elkin's algorithm retained over 820,000 edges, for a stretch of 11.

We will see in the next section how to improve Elkin's algorithm so that it is more effective on roadmaps.

The remainder of this section will work out a small detail that allows Elkin's algorithm (the original version, or the modified version we will present) to a roadmap generation algorithm such as PRM, PRM*, or k-PRM*. Many readers may wish to skip this detail on first reading.

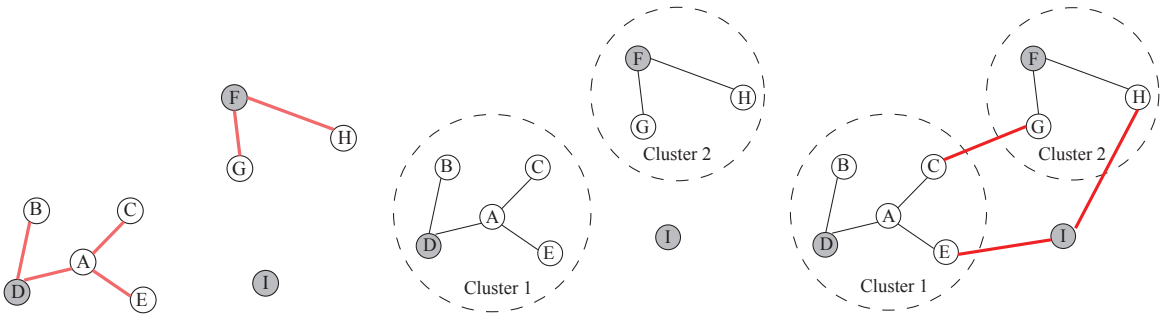
We need to normalize the weight of edges such that 1 will be the minimum edge weight over the graph. The minimum edge weight, however, is hard to estimate before sampling. One way to find this minimum weight is to sample all the vertices before connecting any edge, and check all distances between vertices, without running collision detection. However, in order to admit true online sparse roadmap construction, we would like to avoid this step.

The weighted graph spanner extension guarantees the $(1+\varepsilon)$ approximation as long as on each subgraph, the maximum edge weight is no larger than $(1+\varepsilon)$ times the minimum edge weight. In the following, we will show that without normalizing the minimum edge weight to 1, we can still guarantee each partitioned subgraph satisfies the requirement for a $(1+\varepsilon)$ approximation, as long as all edge weights are positive.

Let us denote the minimum edge weight by w_{min}, and the largest edge weight by w_{max}. Without loss of generality, let us assume $w_{min} < 1$ and $w_{max} > 1$. If we were to normalize the edge weights, the minimum edge weight would be 1 and the maximum edge weight would be w_{max}/w_{min}. Let us denote the number of partitioned subgraph after edge weight normalization as $\hat{\ell}$. Then, we have:

$$\begin{aligned} \hat{\ell} &= \lceil \log_{1+\varepsilon}(w_{max}/w_{min}) \rceil \\ &= \lceil \log_{1+\varepsilon}(w_{max}) - \log_{1+\varepsilon}(w_{min}) \rceil . \end{aligned} \tag{1}$$

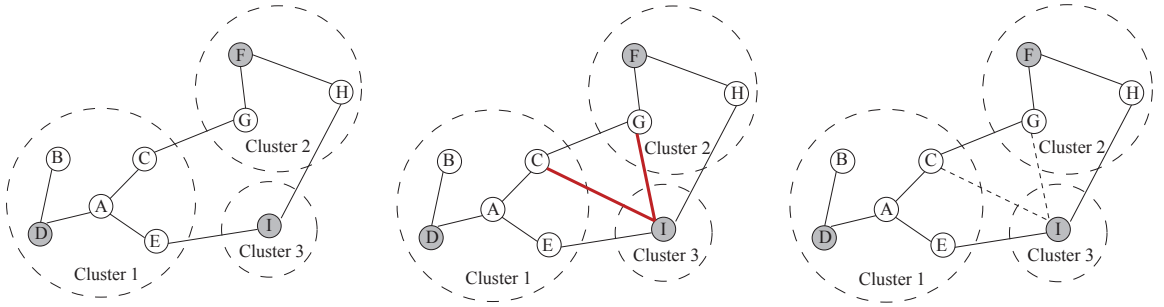
Let us denote the number of partitioned subgraphs without normalizing edge weights as ℓ. We use 1 as a separation line; we partition each edge with $w(e) \geq 1$ into subgraph G_i if $(1+\varepsilon)^{i-1} \leq w(e) < (1+\varepsilon)^i$. We partition edges with weight $w(e)$



(a) Edges (D, B), (D, A), (A, C), (A, E), (F, G), (F, H) arrive.

(b) All edges are retained and are added to clusters. (Vertices D and F are assigned as cluster roots based on order of edge arrival and initial indices of vertices.)

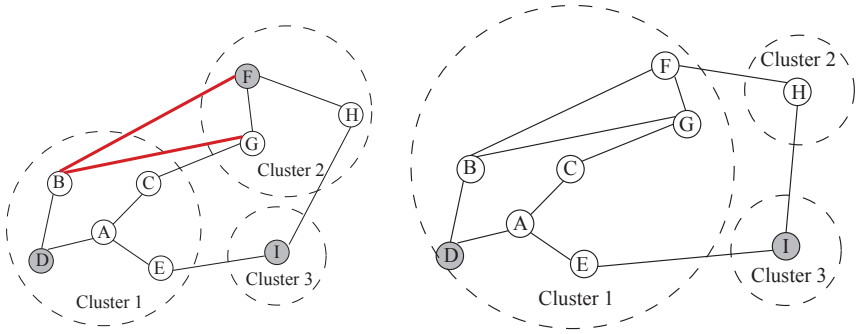
(c) Edges (C, G), (E, I), (H, I) arrive.



(d) Edges are retained as crossing edges since corresponding vertices in the clusters have reached maximum depth.

(e) Edges (C, I), (G, I) arrive.

(f) Edges are discarded since the vertices are the boundary vertices on the cluster, and vertex I has already connected to both clusters.



(g) Edges (B, F), (B, G) arrive.

(h) Edges are stored, and vertices F and G are re-clustered because vertex B is not on the boundary of the cluster 1.

Figure 3: 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.

smaller than 1, into subgraph j if $(1+\varepsilon)^{-j} \leq w(e) < (1+\varepsilon)^{1-j}$. Now, the total number of subgraphs ℓ satisfies

$$\ell = \lceil \log_{1+\varepsilon}(w_{max}) \rceil + \lceil -\log_{1+\varepsilon}(w_{min}) \rceil \ . \tag{2}$$

Lemma 1. *Partitioning the graph without normalizing edge weights will generate at most one additional subgraph.*

Proof. Let us denote the fractional part of a number by $\text{frac}(\cdot)$. For simplicity, let $A = \log_{1+\varepsilon}(w_{max})$ and $B = \log_{1+\varepsilon}(w_{min})$. So, $\hat{\ell} = \lceil A - B \rceil$ and $\ell = \lceil A \rceil + \lceil -B \rceil$.

If $B \geq 0$, then $\text{frac}(A) - \text{frac}(B) < 0$ will lead to $\ell = \hat{\ell} + 1$, otherwise $\ell = \hat{\ell}$. If $B < 0$, then if $\text{frac}(A) - \text{frac}(B) \geq 1$, $\ell = \hat{\ell} + 1$, otherwise $\ell = \hat{\ell}$. □

4 A weighted spanner algorithm

It is easy to conclude that there are redundant edges in the spanners found in previous experiments, based on the fact that there are too many edges in the resulting spanners compared to the spanners found by (the much slower) IRS algorithm. This section explores when and why the redundant edges are kept, and how an algorithm may be designed that avoids keeping redundant edges.

Algorithm 2 will describe an improved method of processing edges; we will refer to the combination of this algorithm with the initialization steps from Elkin's algorithm as the *Weighted Streaming Spanner*, WSS.

4.1 Redundant edges

Consider the example presented in figure 4: a graph with three vertices and three edges, $(A,B) \in G_6$, $(B,C) \in G_8$, and $(A,C) \in G_{10}$. Since all three edges belong to three different subgraphs, each edge will be the only connection in each subgraph between given end points, so all three edges will be retained in Elkin's algorithm. On the other hand, if (A,B) and (B,C) arrive earlier than (A,C), this edge (A,C) could be discarded since $w(A,B) + w(B,C)$ may be smaller than $(2m-1) \cdot w(A,C)$ for reasonably large m.

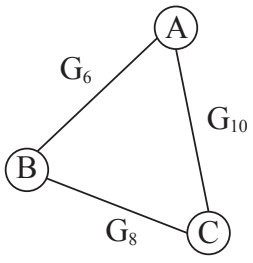


Figure 4: An example of how redundant edges may be stored in the original spanner algorithm for weighted graphs.

In general, the naive extension of Elkin's algorithm to weighted graphs, when processing an edge of a particular subgraph G_i, only considers information generated previously while working on subgraph G_i, even though routes satisfying the stretch may have already been stored using a combination of edges on other subgraphs.

Algorithm 2 shows how some of these redundant edges may be culled. The setup and definitions for this algorithm are the same as for Algorithm 1.

Like Algorithm 1, for each vertex u, a label $P_i(u)$ and a cross-connection list $M_i(u)$ are maintained by the algorithm for each index i, corresponding to subgraph G_i. Similarly, $P_i(v) = I(v)$ and $M_i(v) = \varnothing$ for each i before the execution of the spanner algorithm. All the edge weights are also normalized to lie in the range $[1, \hat{w}]$. For any given edge weight $w(u,v)$, an index $q = 1 + \lfloor \log_{1+\varepsilon} w \rfloor$ is calculated such that edge (u,v) will be partitioned into G_q.

Algorithm 2: PropagatedReadEdge

```
Input: Edge (u,v) with weight w
q ← 1 + ⌊log_{1+ε} w⌋ ;
Let u be the vertex such that P_q(u) ≻ 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) ≻ P_i(v);
        if L(P_i(u)) < (t − 1) then
            P_i(v) ← P_i(u) + n;
    return true;
else if B(P_q(u)) ∉ M_q(v) then
    /* Update cross-connection list:                   */
    M_q(v) ← M_q(v) ∪ {B(P_q(u))};
    return true;
return false;
```

Notice, before storing the edge into the roadmap, collision detection will be called to ensure the edge is valid. Only when the collision detection indicates that the edge is collision free do we store the edge in roadmap and update the corresponding labels and cross-connection lists. If the decision to retain an edge is revoked by collision detection, this does not change the clustering or the result of the spanner algorithm.

Algorithm 2 only broadcasts the labels, but not the cross-connection information out of consideration for simplicity and efficiency. Propagating cross-connection information would require additional processing time as well as additional storage, but would only allow a limited number of redundant edges to be discarded.

The storage space for Algorithm 2 should not increase compared to Algorithm 1 applied to weighted graphs. For each vertex u in each subgraph G_i, since a label $P_i(u)$ and a cross-connection list $M_i(u)$ are maintained in both algorithms, up-

dating the labels does not increase any storage space (detailed discussion of storage space and expected number of edges is in Elkin [2011]), but appending the same edge into multiple cross-connection lists does not happen in Algorithm 1. Therefore, Algorithm 2 only propagates information that does not belong to a cross-connection list and will not increase the expected total storage space.

We will now show that Algorithm 2 still returns a $(1+\varepsilon)(2m-1)$-spanner for input graph G. For this we need a more sophisticated version of the invariant described in Section 3. Before we proceed to the detailed discussion, let us recall that for a given label P of any vertex u, z_C is the unique vertex whose initial value $I(z_C)$ equals the base value of P: $B(P)$. Each vertex u is also related to a randomly assigned integer radius $r(u)$ which is no larger than $m-1$.

Lemma 2. *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_C is at most $L(P)$. Consequently, this distance is at most $r(z_C)$.*

Proof. The latter conclusion follows from the former, because by design (and by definition of "selected label") we have $L(P) \leq r(z_C)$ 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 $z_C = v$ and indeed v is at distance 0 from z_C.

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_C 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_C is at most $L(P)$, completing the proof. □

Theorem 3. *Algorithm 2 applied to a weighted graph generates a $(1+\varepsilon)(2m-1)$-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 $(1+\varepsilon)(2m-1)w(e)$. 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 2, both u and x are at distance at most $r(z_c)$ from z_C. Therefore, following a path from u to z_C to x to v, we see that v is at distance at most $2r(z_c)+1 \leq 2m-1$ 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 $(1+\varepsilon)(2m-1)w(e)$, as required. □

4.2 Runtime for Algorithm 2

One advantage of Algorithm 1 over other streaming spanner algorithms is its constant processing time per edge. Algorithm 2 requires instead *amortized* constant processing time per edge, as we will discuss in this section.

To update one label, the algorithm runs in $O(1)$ time, but since we may propagate this information through all subsequent subgraphs, the label update operation will be called at most ℓ times. Therefore, the processing time per edge is $O(\ell)$ in the worst case (recall that ℓ is the number of subgraphs, which is calculated based on ε chosen by the user, and on the normalized maximum edge weight $\hat{w}$: $\lceil \log_{1+\varepsilon} \hat{w} \rceil \leq \ell \leq \lceil \log_{1+\varepsilon} \hat{w} \rceil + 1)$. Through our experiments, we found that for the environments considered, with roadmaps containing 50,000 vertices, ℓ is less than 50.

Amortized runtime analysis suggests that the runtime should be much less than $O(\ell)$ per edge. Recall that without all the cross connections, the graph is a set of clusters, with a "root" (base vertex) in each cluster. For any vertex belonging to a cluster, its label can be updated at most $m-1$ times, meaning there are at most $m-1$ edges related to this vertex, and each operation will bring another vertex into the cluster. The number of edges in a cluster is linear in the size of the cluster. A collection of clusters with total M vertices contains $O(M)$ edges. Thus the propagate step in Algorithm 2 will be called at most $O(n)$ times. For a graph $G = (V,E)$, to process all edges, at most $|E| + n \cdot \ell$ unit operations will be called (a unit operation could be a update of label, or an insertion into of the cross-connection list). So, amortized processing time per edge is $O(1 + n \cdot \ell / |E|)$. For PRM*, we know that the number of edges $|E| = O(n \log n)$, so the amortized processing time becomes $O(\max\{1, \ell / \log n\})$.

This runtime compares very favorably to the runtime for IRS. Apart from PRM operations (sampling, local planners and collision detections), WSS processes each edge in $O(\max\{1, \ell/\log n\})$, and thus with k-PRM* processes each vertex in $O(\max\{\ell, \log n\}) = O(\max\{\log_{1+\varepsilon} \hat{w}, \log n\})$, since each vertex in k-PRM* roadmaps has $O(\log n)$ neighbors. On the other hand, IRS process each vertex in $O(n \cdot \log^2 n \cdot \log\log n)$ in addition to all the PRM operations.

Practically speaking, as we will see in the next section, the difference in time cost means that while IRS is already fairly expensive for roadmaps with 20,000 vertices, we were able to easily apply WSS to roadmaps with 1.28 million vertices.

5 Experimental results

In order to compare the results of the Weighted Streaming Spanner (WSS) to IRS, we ran both algorithms on several environments from the OMPL (Sucan et al. [December 2012]) motion planning library, using k-PRM* as a base method for generating edges. Results for IRS are based on IRS code from the authors of Marble and Bekris [2012].

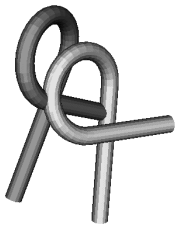


Figure 5: An example of the alpha puzzle environment.

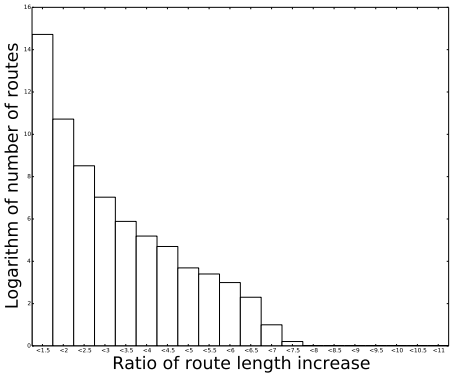


Figure 6: For a roadmap with 50,000 vertices on the Alpha environment with stretch 11, the lengths of paths in the spanner increased compared to those in the k-PRM* by different percentages. The figure shows how many routes, out of a set of routes selected using all-pairs shortest paths among 500 vertices, belong to each different range of length increase.

For simplicity of presentation, we will primarily present experimental results for the *alpha puzzle* environment. A photo of an alpha puzzle is shown in Figure 5. We also applied WSS and IRS to k-PRM* to other environments such as *bugtrap*, *cubicles* and *Easy* from OMPL; the results were consistent with those for the alpha environment.

Table 1 shows comparison results between IRS and WSS, for a stretch factor of 12. Due to run-time considerations, we only applied IRS to roadmaps with up to 40,000 vertices. The experiments were run on a 2014-model laptop, using a single 2.8 GHz core.

5.1 Number of edges retained

Experimentally, IRS retains fewer edges than WSS. This is perhaps due to the more exhaustive approach that IRS takes, in checking new edges against the entire graph to see if deletion will violate the stretch constraint. For example, IRS did quite well on the 20,000-sample experiment in Table 1, retaining only 35,000 out of 510,000 edges. This is close to the theoretical minimum, a spanning tree. WSS retained 120,000 edges in this case. For the graph with 40,000 vertices, IRS retained 190,000 out of 850,000 edges, and WSS retained 280,000.

Qualitatively, we observe that for both spanners, high-stretch paths occur rarely. In order to explore this, we took 500 vertices from each roadmap generated by the spanners, and computed shortest paths between each pair of vertices. Table 1 shows the increase in average path length for these starts and goals.

For example, WSS shows an average path length that is only 1.17 as long as the paths between these samples for k-PRM* for the roadmaps with 20,000 vertices, in spite of the permitted stretch of 12. IRS, presumably because it is more effective at trimming edges, shows a worse average path length ratio of 1.59. For experiments with up to 1.28 million vertices, WSS shows a path length ratio of about 1.28.

Although we might like WSS to be more aggressive at trimming edges, these results for path quality show that application of WSS to a k-PRM may be a very reasonable choice: there are significant benefits in terms of memory and speed, with a tradeoff in path quality that, for much the graph, may not be too bad.

Indeed, it may be possible to modify WSS further to make it more aggressive, but this might come at the cost of the very nice amortized constant processing time per edge. For example, additional book-keeping is almost certainly possible on the crossing edge list that might further reduce the number of edges retained.

What proportion of routes get much worse after application of WSS? Figure 6 shows the distribution for a sample set of runs on the Alpha environment. The vast majority of routes increased by less than a factor of 1.5 (note the logarithmic scale).

5.2 Time costs

The runtime is very different between IRS and WSS. For WSS, regardless of the number of samples used in the roadmap, the expense of collision detection strongly dominates the run-time, since the cost per edge is constant (amortized). On the other hand, for large enough roadmaps, the run-time of IRS is eventually dominated by the spanner algorithm.

In Table 1, we used WSS to filter k-PRM* roadmaps with different number of vertices, ranging from 20,000 to 1.28 million. In all scenarios, WSS was faster than k-PRM*, using less than half of the time to run k-PRM*. IRS, however, is much slower than k-PRM*. We did not run IRS with roadmaps larger than 40,000 vertices, due to computational costs.

5.3 Behavior of WSS for denser graphs

Through the previous experiments, we discovered that the size of spanners returned by the streaming spanner algorithm is

	Time (minutes)			Edges (millions)			Path quality	
samples	k-PRM*	IRS	WSS	k-PRM*	IRS	WSS	IRS	WSS
20000	2.4	372	1.3	0.51	0.035	0.12	1.59	1.17
40000	6.5	2160	3.1	0.85	0.19	0.28	1.64	1.18
80000	13.5	N/A	5.9	2.0	N/A	0.51	N/A	1.24
160000	34.2	N/A	12.8	4.1	N/A	1.0	N/A	1.26
320000	73.2	N/A	32.1	8.3	N/A	2.1	N/A	1.25
640000	162	N/A	58	18	N/A	4.3	N/A	1.28
1280000	258	N/A	102	39	N/A	9.2	N/A	1.28

Table 1: Comparison between IRS and WSS running on k-PRM*. Data reflects averages over 10 runs for k-PRM*, WSS, and IRS with 20,000 vertices. Only a single sample run was performed for each example of IRS with larger numbers of vertices.

greatly affected by both size (number of vertices) and density (number of edges if the number of vertices are fixed) of the original graph. To explore this, we randomly generated roadmaps with different sizes (Figure 7) and density (Figure 8). Each roadmap is considered to be an unweighted graph so we can analyze the behavior of the original streaming spanner algorithm; for weighted graphs, the behavior is similar over each subgraph.

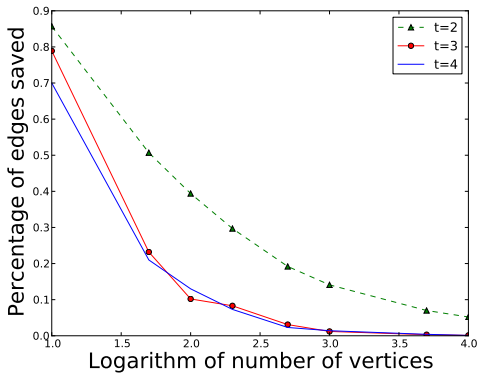


Figure 7: For random roadmaps with different number of vertices, the percentage of edges stored for different k in a $(2k-1)$-spanner using the streaming spanner algorithm.

From figures 7 and 8, it is not hard to conclude that for larger and denser graphs, the streaming spanner algorithm performs better. Therefore, for large roadmaps computed for high-DOF configuration spaces, we expect WSS to still produce roadmaps computationally efficiently while storing an even smaller percentage of edges.

In the extreme case, WSS can even be used to process a complete PRM (which attempts to connect all pairs of samples). Table 2 shows some of the results.

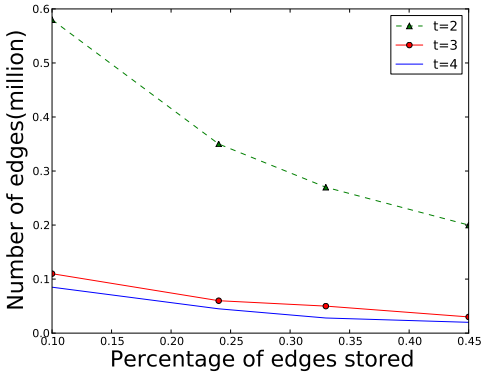


Figure 8: For random roadmaps with 1000 vertices, but different number of edges (density), the percentage of edges stored using the streaming spanner algorithm with different k.

5.4 Approximation parameter ε

In previous experiments, we simply set the approximation parameter $\varepsilon = 0.1$. However, recall that the union of spanners on all subgraphs only $(1+\varepsilon)$ approximates the real spanner on original weighted graph.

This ε affects not only the overall stretch of the spanner, but also affects the graph partitioning, *i.e.*, how many subgraphs we will run the spanner algorithm on. Based on the comparison of running Elkin's algorithm on unweighted graph and weighted graphs, we can hypothesize that the fewer subgraphs we have, the sparser the final spanner will be for the same stretch.

On the same weighted graph with different ε values, the number of subgraphs can be very different. For example, when we tested on the environment Alpha with $\varepsilon = 0.1$, there were about 58 subgraphs, meaning the maximum edge length is about $1.1^{58} \approx 2^8$, so if we set $\varepsilon = 1$, we only have about 8 subgraphs. Therefore, the density should be very different for different ε with same stretch.

What about different ε with different input m such that

	runtime (minutes)	edges retained
Alpha	462.4	0.97%
Apartment	577.9	1.07%
Bugtrap	127.5	0.92%
Cubicles	497.2	0.65%
Easy	317.1	0.73%
Home	434.6	1.01%

Table 2: Application of our modified spanner with stretch $(1+\varepsilon)\times 11$ algorithm to a complete PRM for $50,000$ vertices. The runtime and the percentage of edges retained is shown in the table. Without WSS, the complete PRM requires more than 120 hours to run.

	$(\varepsilon = 0.1, m = 5)$	$(\varepsilon = 1, m = 3)$
relative density	1	0.4
average route length	1.26	1.28
standard deviation	0.18	0.52

Table 3: Comparison of applying WSS with $(\varepsilon = 0.1, m = 5) \to t = 9.9$ and $(\varepsilon = 1, m = 3) \to t = 10$ to 400-nearest neighbors on Alpha environment. The route length is compared to untrimmed 400 neighbor PRM roadmap.

the approximated stretch is roughly the same? We compared between two settings with $(\varepsilon = 0.1, m = 5) \to t = 9.9$ and $(\varepsilon = 1, m = 3) \to t = 10$. It turns out, even when m is smaller for large ε, the overall spanner is still sparser. At the same time, the average route length is quite close. The comparison (applied to 400-nearest neighbor PRM on Alpha environment) result is shown in Table 3.

The large ε leads to smaller spanners, with similar average route lengths compared to the sparse roadmap generated by small ε. However, the standard deviation is much larger for large ε, meaning there are many more long routes on this sparser spanner. For this experiment, the standard deviation implies that only 75% routes are guaranteed to increase their length by no more than 130%.

In the remainder of the text, we will continue to use $\varepsilon = 0.1$ for consistency.

6 Choosing the number of samples: any-time planning with WSS

Elkin's algorithm and WSS require that the number of samples n be known in advance. This number is used to generate the geometric distribution of radii for vertices; if a vertex ends up as the base vertex of a cluster, then its radius determines the maximum distance to any other vertex in the cluster.

Choosing n in advance of running WSS seems problematic. We might like to run the roadmap generation in parallel with queries, and the quality of the roadmap should continue to improve as time and computation become available. This section presents two approaches to allow continuous improvement of the roadmap.

6.1 Iterative WSS

The first approach to allowing the size of the roadmap to increase with time is based on the idea of Iterative Deepening Depth-First Search (IDDFS) (Korf [1985]). Choose a small initial number of samples, n. Run WSS and return the spanner. If more time or computation is available, double n, re-sample, and use WSS to compute the next spanner. Repeat.

This approach might seem to be quite expensive, but just as with IDDFS, the dominant factor in the overall run-time will the be the last iteration, due to the doubling in size of n at each step. We tested the algorithm on the *Alpha* environment; unsurprisingly, the run-time of iterative WSS was less than twice as long as that of the non-iterative approach for any n used as a stopping point.

6.2 Simplified WSS

There is a simpler approach to allowing WSS to continuously improve results: just choose a very large value for n, and whenever a query arrives, use the roadmap that has been constructed to that point. Since the spanner property is invariant over iterations of WSS, any intermediate roadmap is a spanner over the edges that have arrived so far.

From a theoretical standpoint, this approach is problematic – Elkin's proofs of the expected number of edges in a spanner rely on the geometric distribution of radii, and the geometric distribution for a very large value of n is different than that which would be constructed for a smaller n. However, perhaps surprisingly, our experimental results appear to show that the intermediate spanners generated in this way are essentially just as good as if the "correct" value of n had been chosen.

We simplified Elkin's algorithm even further, and instead of choosing a geometric distribution for the radii, we simply chose every radius to be $m-1$, based only on the stretch. Experimentally, the quality of intermediate spanners did not degrade; Table 4 shows the spanner size (number of edges), and Table 5 shows the path quality comparison.

Why is the experimental performance of this simplified algorithm (with very large n and uniform radii) just as good as that of the original? In the original algorithm, the ultimate result is that most clusters are of size $m-1$; vertices with small radii are eventually re-assigned into larger clusters. Since the vertices with large radii were chosen at random, there is nothing particularly special about them. In the simplified algorithm, any vertex may end up as the center of a cluster of size $m-1$. Apparently, cluster formation still leads to good properties of the resulting spanner. Other than the geometric distribution, the assignment of vertices to clusters is independent of

Stretch	3.3	5.5	7.7	9.9	12.1
WSS	20.7%	7.68%	4.99%	4.76%	3.96%
SWSS	20.3%	6.87%	4.69%	4.58%	3.72%

Table 4: Number of edges stored in Alpha environment using WSS and Simplified WSS for different stretch.

Stretch	3.3	5.5	7.7	9.9	12.1
Ratio	1.006	1.09	0.98	1.02	1.03

Table 5: The average route length of Simplified WSS roadmaps over WSS roadmaps.

the value of n, so long as n is sufficiently large.

In spite of the lack of rigorous theoretical justification for the expected size of the simplified spanner (which still correctly maintains the stretch) relative to iterative WSS, from a practical perspective, this simplified WSS is easier to implement, faster, and in our experiments, gave comparable results.

7 An online directed spanner algorithm

Some robotic systems require the use of directed graphs as roadmaps. In particular, if there exist configurations A and B such that the cost of moving from A to B is different than the cost of moving from B to A, a directed graph is required. Such asymmetric systems particularly arise if there are asymmetric non-holonomic constraints (for example, consider the Dubins car, see Dubins [1957]). We restrict consideration in this section to asymmetric systems for which a local planner can be provided.

Finding directed spanners effectively is still an open problem, and (offline) Integer-Linear Programming (ILP) is still the major technique used to find directed spanners (for example, see Bhattacharyya and Makarychev [2010]).

In this section, we will present an online algorithm for finding directed spanners. We prove that the algorithm is correct (maintains the stretch). We also show that for any directed graph, there exist certain orders of edge arrivals such that *no* online spanner will significantly reduce the number of edges. The existence of bad sequences of edges limits what can be proven about the space requirements of our algorithm, or indeed about any online spanner algorithm. However, such bad sequences of edge arrivals appear to happen only infrequently. Experimentally, we show that the online directed spanner we present typically stores $20\% - 32\%$ of edges for k-PRM* roadmaps with $20,000$ vertices, with stretch of 12.1.

7.1 Limitations of online directed spanner algorithms

For an online undirected spanner algorithm, an upper bound on the distance between a pair of vertices is known as soon as an edge arrives. However, for a directed graph, there may initially be no useful upper bound on the return distance $B \to A$ at the time the edge $A \to B$ arrives. (For example, the local planner from $B \to A$ may not find a path, even though an as-yet-unknown longer route through other vertices does exist.) This leads to fundamental difficulties in constructing a "good" online directed spanner, as we will now prove.

Lemma 4. *There exists a graph S_n and sequences of edge arrivals such that no online directed spanner algorithm can discard any edge, but an offline directed spanner algorithm can discard some of the edges.*

The idea of the proof is to construct a sequence of edges of a completely connected directed graph such that as each edge arrives, it forms the first connection between its vertices. Since the structure of the entire graph is not known a priori by the algorithm, each such edge must be stored. Mechanically, to find graphs with "bad" orders of edge arrivals, the proof will construct a sequence of subgraphs such that all of the edges of earlier subgraphs in the sequence arrive before edges in later subgraphs. Since the edges of later subgraphs are not known a priori, certain edges in earlier subgraphs cannot be discarded without risking violating the stretch.

Proof. We construct the graph S_n as a subgraph of a complete connected directed graph G, with vertices $\{v_1, v_2, \ldots, v_n\}$. Let S_1 be the graph of all incoming edges of v_1. For $i > 1$, let S_i contain S_{i-1} and all the incoming edges of v_i except the edges from $\{v_1, v_2, \ldots, v_{i-1}\}$ to v_i.

Let all the edges of S_i arrive earlier in the stream than edges in $S_{i+1} - S_i$. No online directed spanner algorithm can discard any edge until S_n has been completely constructed. In S_i, all the incoming edges for v_i are processed except those from $\{v_1, v_2, \ldots, v_{i-1}\}$. Any arriving edge $(v_k \to v_i)$ with $k > i$ is the first connecting route from v_k to v_i, and must be stored.

However, there are redundant edges in S_n that can be discarded. For any $i \in [1, n-2]$ there exists a triangle of edges $v_{i+1} \to v_{i-1}$, $v_i \to v_{i-1}$, and $v_{i+1} \to v_i$. For any spanner with stretch larger than or equal to 2, one can discard edge $(v_{i+1} \to v_{i-1})$ once S_n is known. (Once S_n is known, it is clear that there are no missing edges that require $(v_{i+1} \to v_{i-1})$ to maintain connectivity.) □

We will now show that bad orders of edge arrivals can occur for *any* graph.

Theorem 5. *For any directed graph G, there exists a subgraph S_n which contains at least 50% of the edges of G. If the edges of S_n are streamed in some certain order, no online directed spanner algorithm can discard any edge in S_n.*

Proof. Select S_n to be the largest acyclic subgraph of G; it is well-known that S_n contains at least half of the edges of G. Sort the vertices in reverse topological order; any edge from u to v will be the first connecting route from u to v; therefore no online spanner algorithm can discard any edge of S_n, since it might be required to maintain the stretch if some as-yet-unknown shorter route exists from u to v. □

The previous proof shows that there is at least one way of choosing a bad edge ordering for any graph. In fact, there are typically many vertex orderings that allow bad edge orderings to be found. To demonstrate this, we now describe another (perhaps more direct) method for constructing an acyclic subgraph S_n that contains at least half of the edges of G.

Denote the in-degree of v by $\text{In}(v)$, and denote the out-degree of v by $\text{Out}(v)$. Set $S_0 = \emptyset$, $T_1 = G$. Starting from $i = 1$, $T_i = (V_i, E_i)$, at each iteration, find the $k = \text{argmax}_{v_j \in V_i} |\text{In}(v_j) - \text{Out}(v_j)|$. Let S_i be the union of S_{i-1} and all the incoming edge of v_k in T_i, T_{i+1} equal T_i without v_k and all v_k's incoming and outgoing edges.

At each step, S_i is acyclic. At step i with selected vertex v_k, any incoming edge of v_k in T_i $u \to v_k$ is the first connection from u to v_k. Therefore, no online directed spanner algorithm can discard any edge in S_n following the given sequence.

At each step i, we know for a fact that in T_i, $\sum_{v_j \in V_i} \text{In}(v_j) = \sum_{v_j \in V_i} \text{Out}(v_j)$. Therefore, for $k = \text{argmax}_{v_j \in V_i} |\text{In}(v_j) - \text{Out}(v_j)|$, $\text{In}(v_k) - \text{Out}(v_k) \geq 0$. Therefore, at each step i, if l edges are stored into S_i, then $|E_i| - |E_{i+1}| \leq 2l$. So at least half of the edges of G will be in S_n.

There exist other sequences that could let S_n contain at least 50% edges, while no edges of in S_n can be discarded by online spanner algorithms. For example, where the previous vertex order sorted vertices by in degree minus out degree, reversing the order also gives an acyclic subgraph containing at least half the edges. Even worse, given a vertex ordering, re-ordering how edges arrive at each vertex also gives a bad edge arrival sequence.

7.2 An online directed spanner algorithm

Despite all the potential problems that online directed spanner algorithms face if the ordering of edges is unfortunate, we do not see any practical alternatives that can quickly find sparse spanners for directed PRM* roadmaps. For example, IRS could be applied with only minor modifications, but is much slower than we would like, and also provides no guarantees about the number of edges that might be discarded.

In this section, we present an online directed spanner algorithm that is provably correct, experimentally fast, and discards many edges. Our algorithm extends Elkin's algorithm, but maintains additional book-keeping information to ensure correctness on directed graphs.

Consider the behavior of Algorithm 1 as applied to vertices u and v in the same cluster. For an undirected graphs, u and v are connected, and the path between them contains no more than $2m-2$ edges. However, following the same cluster construction method for a directed graph, there may not be a path from u to v, or from v to u, or both.

One way to allow u to connect to every other vertex in the cluster while maintaining the stretch is to ensure that u can reach the base vertex of this cluster using fewer than m edges. Therefore, in addition to the label and the cross-connection list, another list $Rt(u)$ may be kept for each vertex u, initially empty, to keep track all the base vertices u is connected to.

In Algorithm 1, $M(u)$ is a list. But in the online directed spanner algorithm (Algorithm 3), $M(u)$ is a map, such that each time we add an edge $(u \to v)$ into the cross-connection map, a pair $(B(P(v)), I(v))$ is inserted into $M(u)$. These intermediate vertices can be updated to avoid over-connecting between clusters.

Algorithm 3: Online directed spanner

Input: $(u \to v)$, m, $n = |V|$
if $L(P(u)) < r(B(P(u)))$ **then**
  if *u is* ***not*** *root* ***and*** $Rt(u)[B(P(v))] \in (0,k]$ **then**
    return **false**;
  if *u is root* **then**
    $Rt(u) \leftarrow Rt(u) \cup (B(P(u),0))$;
  if *v is root* ***and*** $Rt(u)[B(P(u))] < k$ **then**
    $Rt(u) \leftarrow Rt(u) \cup (B(P(v)), Rt(v)[B(P(u))]+1)$;
  $P(v) \leftarrow P(u)+n$;
  return **true**;
else if $Rt(u)[B(P(v))] \leq k$ **then**
  return **false**;
else if *u and v belong to same cluster* ***and*** $Rt(v)[B(P(v))] < k$ **then**
  $Rt(u) \leftarrow Rt(u) \cup (B(P(v)), Rt(v)[B(P(v))]+1)$;
  return **true**;
else if $B(P(v)) \notin M(u)$ **then**
  $M(u) \leftarrow M(u) \cup (B(P(v)), I(v))$;
  return **true**;
else if $Rt(M(u)[B(P(v))])[B(P(v))] < k$ **then**
  return **false**;
else if $Rt(v)[B(P(v))] < k$ **then**
  $M(u)[B(P(v))] = I(v)$;
  return **true**;

Theorem 6. *Algorithm 3 will return a* $(2m-1)$*-spanner for any directed graph G.*

Proof. For any two vertices u and v, if the edge $(u \to v)$ is not stored, this is because either u has been connected to the base vertex of v, and the distance is less than m, or because u has been connected to the cluster containing v through w, and w can get to the base vertex of that cluster within $m-1$ steps. In

stretch	5.5	7.7	9.9	12.1
5 polygons	37.2%	29.3%	24.5%	20.6%
10 polygons	53.3%	42.5%	33.1%	27.8%
20 polygons	66.9%	51.1%	39.6%	32.21%

Table 6: Percentages of edges stored for a directed roadmap generated in a 16×16 planar field with random polygonal obstacles. Each roadmap contains $20,000$ vertices.

either case, there are at most $2m-1$ edges from u to v, so the stretch is maintained. □

This algorithm can also be extended to weighted graphs, as described in section 2.1. However, note that this algorithm cannot find spanners for $m = 2$ efficiently. When $m = 2$, each cluster has maximum radius 1, and each vertex needs to find a root to get to other vertices in this algorithm. Therefore, many edges are stored by this algorithm if $m = 2$.

The runtime of this algorithm can be analyzed as follows. For each edge, the query need only find if an element exists in a set. By using a hash-like data structure (the same one used in Elkin's original algorithm) to maintain the cross-connection list and the root list, the algorithm is essentially constant time per edge on unweighted graph.

8 Planning for Dubins' car among polygonal obstacles

To demonstrate the directed spanner algorithm, we consider the following very simple non-holonomic system: Dubins' car in a plane with random polygons as obstacles. The car has one control, ω– the angular velocity of the car. The state space has three dimensions; a state q can be represented as $q = (x, y, \theta)^T$.

The state equations are:

$$\dot{q} = \begin{pmatrix} \cos(\theta(t)) \\ \sin(\theta(t)) \\ \omega \end{pmatrix}. \quad (3)$$

We use an optimal path finder for Dubins' car as local planner. Since the configuration space is only 3-dimensional, only $20,000$ samples were generated for each roadmap. We used the k-PRM* connecting strategy.

The directed spanner can trim many edges for roadmaps constructed, but as the environment becomes more and more complex due to the addition of more obstacles, the spanner can trim fewer edges. Some experimental results are shown in Table 6. Figure 9 shows a few selected routes generated by k-PRM*, and routes after applying spanner algorithm.

We also applied the algorithm to random directed graphs, and the spanner algorithm consistently removes over 60 to 80 percent edges for different stretches.

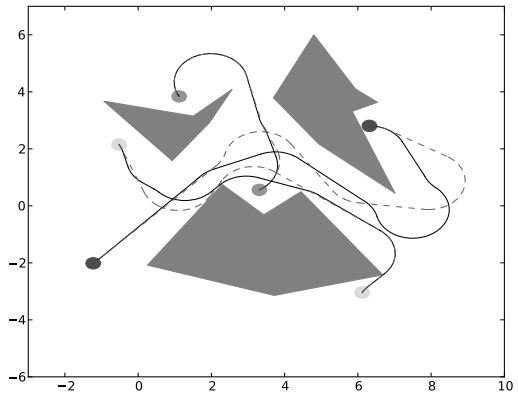


Figure 9: PRM* paths of Dubins' car (solid), and the paths found by spanner (dashed).

9 Conclusions and Acknowledgments

In this work, we introduced a fast state-of-art streaming spanner algorithm from the graph theory community. The algorithm was not originally suitable for roadmaps (due to the large number of edges stored), but our modified algorithm that propagates information stores many fewer edges. A simplified version and iterative version of WSS were further proposed to allow any-time planning.

We then extended the algorithm to directed graphs, and presented a provably-correct online directed spanner algorithm. Experiments shows that the algorithm works well in practice even though no theoretical bounds on the number of edges stored can be provided.

The authors would like to acknowledge funding support from the Dartmouth Neukom Institute for Computational Sciences, and from NSF grant IIS-1451632.

References

Nancy M. Amato, O. Burchan Bayazit, Lucia K. Dale, Christopher Jones, and Daniel Vallejo. OBPRM: an obstacle-based PRM for 3D workspaces. In *Proceedings of the third workshop on the algorithmic foundations of robotics on Robotics*, WAFR '98, pages 155–168, Natick, MA, USA, 1998. A. K. Peters, Ltd. ISBN 1-56881-081-4.

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

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

Arnab Bhattacharyya and Konstantin Makarychev. Improved approximation for the directed spanner problem. *Computing Research Repository, CoRR*, abs/1012.4062, 2010.

Prosenjit Bose, Vida Dujmovic, Pat Morin, and Michiel H. M. Smid. Robust geometric spanners. *Computing Research Repository, CoRR*, abs/1204.4679, 2012.

John F. Canny. *The Complexity of Robot Motion Planning*. MIT Press, Cambridge, MA, 1988.

Edith Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM J. Comput.*, 28(1):210–236, 1998.

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

Lester E. Dubins. On curves of minimal length with a constraint on average curvature and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, 79:497–516, 1957.

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. ISSN 1549-6325. doi: 10.1145/1921659.1921666.

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. ISBN 0-89871-585-7.

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.

David Hsu, Robert Kindel, Jean-Claude Latombe, and Stephen M. Rock. Randomized kinodynamic motion planning with moving obstacles. *I. J. Robotic Res.*, 21(3):233–256, 2002.

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

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.

Lydia E. Kavraki, M. N. Kolountzakis, and Jean-Claude Latombe. Analysis of probabilistic roadmaps for path planning. *IEEE Transactions on Robotics and Automation*, 14 (1):166–171, 1998.

Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, September 1985. ISSN 0004-3702. doi: 10.1016/0004-3702(85)90084-0.

Anthanasios Krontiis, Andrew Dobson, and Kostas Bekris. Sparse roadmap spanners. *Proceedings of the workshop on the algorithmic foundations of robotics on Robotics, WAFR 2012*, 2012.

Jean-Paul Laumond and Carole Nissoux. Visibility-based probabilistic roadmaps for motion planning. *Journal of Advanced Robotics*, 14:2000, 2000.

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

Jyh-Ming Lien, Shawna L. Thomas, and Nancy M. Amato. A general framework for sampling on the medial axis of the free space. In *ICRA*, pages 4439–4444. IEEE, 2003.

Tomás Lozano-Pérez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, C-32:108–120, 1983.

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

James D. Marble and Kostas 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 2011b.

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

James D. Marble and Kostas E. Bekris. Asymptotically near-optimal planning with probabilistic roadmap spanners. *IEEE Transactions on Robotics*, 29(2):432–444, 2013.

David Peleg and Alejandro A. Schffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989. ISSN 1097-0118. doi: 10.1002/jgt.3190130114.

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. ISSN 1549-6325. doi: 10.1145/1367064.1367069.

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

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.

Doron Shaharabani, Oren Salzman, Pankaj K. Agarwal, and Dan Halperin. Sparsication of motion-planning roadmaps by edge contraction. In *ICRA*. IEEE, 2013.

Ioan A. Sucan, Mark Moll, and Lydia E. Kavraki. The open motion planning library. *IEEE Robotics and Automation Magazine*, 19(4):72082, December 2012. URL `http://ompl.kavrakilab.org`.

Petr Svestka and Mark H. Overmars. Motion planning for carlike robots using a probabilistic learning approach. *I. J. Robotic Res.*, 16(2):119–143, 1997.

Mikkel Thorup and Uri Zwick. Approximate distance oracles. In Jeffrey Scott Vitter, Paul G. Spirakis, and Mihalis Yannakakis, editors, *ACM Symposium on the Theory of Computing, STOC 2001*, pages 183–192. ACM, 2001. ISBN 1-58113-349-9.

Weifu Wang, Devin J. Balkcom, and Amit C. Chakrabarti. A fast streaming spanner algorithm for incrementally constructing sparse roadmaps. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1257–1263, 2013.