Practical Construction of Metric t-Spanners *

Gonzalo Navarro and Rodrigo Paredes

Center for Web Research, Dept. of Computer Science, University of Chile.
Blanco Encalada 2120, Santiago, Chile.
{gnavarro,raparede}@dcc.uchile.cl

Abstract. Let $G(V, A)$ be a connected graph with a nonnegative cost function $d : A \to \mathbb{R}^+$. Let $d_G(u, v)$ be the cost of the cheapest path between $u, v \in V$. A t-spanner of G is a subgraph $G'(V, E)$, $E \subseteq A$, such that $\forall\ u, v \in V,\ d_{G'}(u, v) \leq t \cdot d_G(u, v),\ t > 1$. We focus on the metric space context, which means that $A = V \times V$, d is a metric, and $t \leq 2$. Several algorithms to build t-spanners are known, but they do not apply well to our case. We present four practical algorithms to build t-spanners with empirical $O(n^{2.24})$ time cost and $O(n^{1.13})$ edges. These algorithms are useful on general graphs as well.

1 Introduction

Let G be a connected graph $G(V, A)$ with a nonnegative cost function $d(e)$ assigned to its edges $e \in A$. The shortest path among every pair of vertices $u, v \in V$ is the one minimizing the sum of the cost of the edges traversed. This can be computed with Floyd's algorithm or with $|V|$ iterations of Dijkstra's algorithm considering each vertex as the origin node [16]. A t-spanner it is a subgraph $G'(V, E)$, with $E \subseteq A$, which permits to compute paths with *stretch* t, that is, ensuring that $\forall u, v \in V, d_{G'}(u, v) \leq t \cdot d_G(u, v)$ [12]. We call this the t-condition.

In this work we are interested in using t-spanners as tools for searching metric spaces [5]. A metric space is a set of objects $\mathbb{X}$ and a distance function d defined among objects, which satisfies the metric properties (positiveness, reflexivity, symmetry, triangle inequality). Given a finite subset $\mathbb{U} \subseteq \mathbb{X}$, of size n, the goal is to build a data structure over $\mathbb{U}$ such that later, given a query object $q \in \mathbb{X}$, one can find the elements of $\mathbb{U}$ close to q with as few distance computations as possible.

One of the best existing algorithms to search metric spaces is AESA [15]. AESA precomputes and stores the matrix of $n(n-1)/2$ distances among elements of $\mathbb{U}$. This huge space requirement makes it unsuitable for most applications, however.

This matrix can be seen as a complete graph $G(V, A)$ where the set of vertices $V = \mathbb{U}$ corresponds to the objects of the metric space, and the set of edges A corresponds to the $n(n-1)/2$ distances among these objects. A t-spanner G' of G would represent all these distances using a small number of edges $E, E \subseteq A$, and still would be able to approximate all the distances with a maximum error t, that is:

$$d(u, v) \quad \leq \quad d_{G'}(u, v) \quad \leq \quad t \cdot d(u, v) \tag{1}$$

In most metric spaces the distance histogram follows a distribution that becomes concentrated as the dimension increases [5]. This means that in practice we are interested in the range $t \in (1, 2]$.

We pursue this line in [11], where we focus on the search process but not on t-spanner construction. Hence our interest in this paper is in building t-spanners over metric spaces which work well in practice. Few algorithms exist apart from the basic $O(mn^2)$ technique ($m = |E|$), which inserts the edges needed one by one and recomputes all the shortest paths to every edge inserted.

Four t-spanner construction algorithms are presented in this paper, with the goals of decreasing CPU and memory cost and of producing t-spanners of good quality, i.e., with few edges. Our four algorithms are:

* This work has been supported in part by the Millenium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile (both authors), CYTED VII.19 RIBIDI Project (both authors) and AT&T LA Chile (2nd author).

1. An optimized basic algorithm, where we limit the propagation of an edge insertions.
2. A massive edge insertion algorithm, where we amortize the cost of recomputing distances among many edge insertion.
3. An incremental algorithm, where nodes are added one by one to a correct t-spanner.
4. A recursive algorithm applying a divide and conquer technique.

Table 1 shows the complexities obtained. We obtain empirical $O(n^{2.24})$ time cost and $O(n^{1.13})$ edges. This shows that good quality t-spanners can be built in reasonable time (just the minimum spanning tree computation needs $O(n^2)$ time). We take no particular advantage of the metric properties of the edge weights, so our algorithms can be used on general graphs too.

	Basic	Basic optimized	Massive edge insertion	Incremental	Recursive
CPU time	$O(mn^2)$	$O(mk^2)$	$O(nm \log m)$	$O(nm \log m)$	$O(nm \log m)$
Memory	$O(n^2)$	$O(n^2)$	$O(m)$	$O(m)$	$O(m)$
Distance evaluations	$O(n^2)$	$O(n^2)$	$O(nm)$	$O(n^2)$	$O(n^2)$

Table 1. t-Spanner algorithm complexities comparison. The value k refers to the number of nodes that have to be checked when updating distances due to a new inserted edge.

2 Previous Work

Several studies on general graph t-spanners have been undertaken [7, 12, 13]. Most of them resort to the naive $O(mn^2)$ time construction approach detailed in the next section, where $n = |V|$ and $m = |E|$ refer to the resulting t-spanner. It was shown in [1, 2] that this technique produces spanners with $n^{1+O(\frac{1}{t-1})}$ edges on general graphs of n nodes. This result, however, is not interesting for $t \leq 2$.

More sophisticated algorithms have been proposed in [6], producing t-spanners with $O(n^{1+(2+\varepsilon)(1+\log_n m)/t})$ edges in time $O(mn^{(2+\varepsilon)(1+\log_n m)/t})$, where in this case m refers to the original graph. In a metric space $m = \Theta(n^2)$, which means that we need time $O(n^5)$ at least. Additionally, the algorithms in [6] work for $t \in [2, \log n]$, unsuitable for our application. Parallel algorithms have been pursued in [10], but they do not give new sequential algorithms.

As it can be seen, none of these results is useful for our problem.

As it regards to Euclidean spanners, i.e., the subclass of metric spanners where the objects are points in a D-dimensional space with Euclidean distance, much better results exist [7, 1, 2, 9, 8, 14], showing that one can build t-spanners with $O(n)$ edges in $O(n \log^{D-1} n)$ time. These results, unfortunately, make heavy use of coordinate information and cannot be extended to general metric spaces.

Other related results refer to probabilistic approximations of metric spaces using tree metrics [3, 4]. The idea is to build a set of trees such that their union makes up a t-spanner with high probability. However, the t values are of the form $O(\log n \log \log n)$.

Hence the need to find algorithms that allow building appropriate t-spanners for metric spaces, that is, with $t \leq 2$, for complete graphs, and taking advantage of the triangle inequality.

3 Basic t-Spanner Construction Algorithm

The intuitive idea to solve this problem is iterative. We begin with an initial t-spanner that contains all the vertices and no edges, and calculate the distance estimations among all vertex pairs. These are all infinite

at step 0, except for the distances between a node and itself ($d(u,u) = 0$). The edges are then inserted until all the distance estimations fulfill the t-condition.

The edges are considered in ascending cost order, so we start by sorting them. Using smaller-cost edges first is in agreement with the geometric idea of inserting edges between near neighbors and making up paths from low cost edges in order to use few edges overall.

Hence the algorithm uses two matrices. The first, *real*, contains the true distance between all the objects, and the second, *estim*, contains the distance estimations obtained with the t-spanner under construction. The t-spanner is stored in an adjacency list.

The insertion criterion is that an edge is added to the set E only when its current estimation does not satisfy the t-condition. After inserting the edge, it is necessary to update *all* the distance estimations. The update mechanism is similar to the distance calculation mechanism of Floyd's algorithm, but considering that edges, not nodes, are inserted into the set. Figure 1 depicts the basic t-spanner construction algorithm.

```
t-Spanner0 (Stretch t, Vertices 𝕌)

real ← real distance matrix
estim ← estimated distance matrix
t-Spanner ← t-spanner edge structure // initially ∅

for e = (e_u, e_v) ∈ real chosen in increasing cost order do
   if estim(e) > t · real(e) // e is not well t-estimated
      t-Spanner ← t-Spanner ∪ {e}
      for v_i, v_j ∈ 𝕌
         d_1 ← estim(v_i, e_u) + estim(v_j, e_v)
         d_2 ← estim(v_j, e_u) + estim(v_i, e_v)
         estim(v_i, v_j) ← min(estim(v_i, v_j), min(d_1, d_2)+real(e))
```

Fig. 1. Basic t-spanner construction algorithmi (t-Spanner 0).

This algorithm makes $O(n^2)$ distance evaluations, like AESA [15]; $O(mn^2)$ CPU time (recall that $n = |V|$ and $m = |E|$); and $O(n^2 + m) = O(n^2)$ memory. Its main deficiencies are excessive edge insertion cost and too high memory requirements.

4 Optimized Basic Algorithm

Like the basic algorithm (section 3), this algorithm considers the use of *real* and *estim* matrices, choosing the edges in increasing weight order. The optimization focuses on the distance estimation update mechanism.

The main idea is to control the propagation of the computation, that is, only updating the distance estimations that are affected by the insertion of a new edge. Figure 2 shows the insertion of a new edge. In the first update we must modify only the edge that was inserted, between nodes a_1 and a_2. The computation then propagates to the neighbors of the a_i nodes, namely the nodes $\{b_1, b_2, b_3\}$; then to the nodes $\{c_1, c_2\}$ and finally d_1. The propagation stops when a node does not improve its current estimation or when it does not have further neighbors.

In order to control the propagation, the algorithm uses two sets, *ok* and *check*.

- *ok*: The nodes that already have updated their shortest path estimations due to the inserted edge.
- *check*: The adjacency of *ok*, $check = adyacency(ok) - ok = \{u \in \mathbb{U}, \exists v \in ok, (u,v) \in E\} - ok$. These are the nodes that we still need to update.

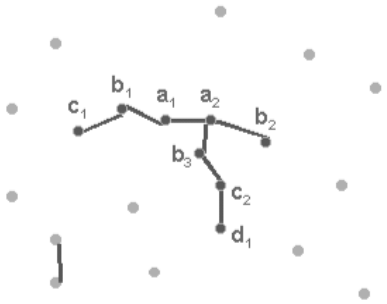

Fig. 2. Propagation of distance estimations.

Note that it is necessary to propagate the computation only to the nodes that improve their estimation to a_1 or a_2. The complete algorithm reviews all the edges of the graph. For each edge, it iterates until no further propagation is necessary. Figure 3 depicts the optimized basic algorithm.

```
t-Spanner1 (Stretch t, Vertices 𝕌)

real ← real distance matrix
estim ← estimated distance matrix
t-Spanner ← t-spanner edge structure // initially ∅

for e = (e_u, e_v) ∈ real chosen in increasing cost order do
   if estim(e) > t · real(e) // e is not well t-estimated
      t-Spanner ← t-Spanner ∪ {e}
      ok ← {e_u, e_v}
      check ← adjacency(ok) − ok
      for c ∈ check
         if ((estim(c, e_v) + real(e) ≤ estim(c, e_u) or (estim(c, e_u) + real(e) ≤ estim(c, e_v))
            for o ∈ ok
               d_1 ← estim(c, e_u) + estim(o, e_v)
               d_2 ← estim(c, e_v) + estim(o, e_u)
               estim(c, o) ← min (estim(c, o), min(d_1, d_2) + real(e))
            check ← check ∪ (adjacency(c) - ok)
         ok ← ok ∪ {c}
         check ← check − {c}
```

Fig. 3. Optimized basic algorithm (t-Spanner1).

This algorithm takes $O(n^2)$ distances evaluations. In terms of CPU time it takes $O(mk^2)$, where k is the number of neighbors to check when inserting an edge. In the worst case this becomes $O(mn^2)$ just like the basic algorithm, but the average is much better. From the point of view of the memory it still takes $O(n^2 + m) = O(n^2)$. This algorithm reduces the CPU time used, but even so this is still very high, and the memory requirements are still too high.

A good feature of this algorithm is that it produces good-quality spanners (few edges), just like the basic algorithm. So we have used its results to predict the expected number of edges per node in order to speed up other algorithms that rely on massive edge insertion. We call $E_{t-\mathrm{Spanner1}}(n, d, t)$ the expected number of

edges in a metric space of n objects, distance function d, and stretch t. In section 8 we show some estimations obtained, see Eq. (2).

5 Massive Edges Insertion Algorithm

This algorithm tries to reduce both the CPU processing time and memory requirements. To reduce the CPU time, the algorithm updates the distance estimations only after performing many edge insertions, using an $O(m \log n)$-time Dijkstra's algorithm to update distances. To reduce the memory requirement, it computes the distances between objects on the fly.

Since we insert edges less carefully than before, the resulting spanner is necessarily of lower quality. Our effort is in minimizing this effect.

The algorithm has three stages. In the first one, it builds the t-spanner backbone by inserting whole minimum spanning trees (MSTs), and determines the global wrongly t-estimated edge list (*pending*); in the second one, it refines the t-spanner by adding more edges to improve the wrongly t-estimated edges; and in the third one, it inserts all the remaining "hard" edges.

This algorithm uses two heuristic values:

H_1 determines the expected number of edges per node, and it is obtained from the t-Spanner1 edge model: $H_1 = |E_{t-\text{Spanner1}}(n, d, t)|/n$. With H_1 we will define thresholds to determine whether or not to insert the remaining edges (those still wrongly t-estimated) of the current node.

H_2 is used to determine the *pending* list size and will give a criterion to determine when to insert an additional MST. The maximum *pending* list size is $H_2 = 1.2 \cdot |E|$, where E refers to the t-spanner under construction.

The algorithm stages are:

1. We insert successive MSTs to the t-spanner. The first MST follows the basics Prim algorithm [16], but the next MSTs are built using Prim over the edges that have not been inserted yet.
   We traverse the nodes sequentially, building the list of pending edges (wrongly t-estimated). At the same time, we insert successive MSTs and remove pending edges accordingly. Additionally, when the current node has no more than $H_1/2$ pending edges, we just insert them. The insertion of MSTs continues as long as there are more than H_2 pending edges (note that H_2 depends on the current t-spanner size $|E|$). This stage continues until we review all the nodes. The output is the t-spanner backbone (*t-Spanner*) and the gobal list of pending edges (*pending*).
2. In the second stage we reduce the *pending* list. For this sake, we traverse the list of nodes with pending edges (*pendingNodes*), from more to less pending edges. For each such node, we check which edges have to improve their t-estimation and which do not (edges originally in the pending list may have become well t-estimated along the process). From the still wrongly t-estimated edges, we insert the $H_1/4$ smaller cost edges and proceed to the next node.
   This allows us to review in the first place the nodes that require more attention, without concentrating all the efforts in the same node.
   The process considers two special cases. The first one is that we have inserted more than n edges, in which case we regenerate and re-sort the *pendingNodes* list and restart the process. The second one is that the pending list of the current node is so small that we simply insert its elements.
   The output condition of the second stage is that the *pending* list size is smaller than $n/2$.
3. We insert the *pending* list to the t-spanner.

Figure 4 depicts the massive edges insertion algorithm. This algorithm takes $O(nm)$ distance evaluations, $O(nm \log m)$ CPU time (since we run Dijkstra's algorithm once per node), and $O(n + m) = O(m)$ memory. It is easy to see that the space requirement is $O(m)$: the *pending* list is never larger than $O(m)$ because at

each iteration of stage 1 it grows at most by n, and if it becomes larger than $1.2 \cdot m$ we take out n edges from it by adding a new MST. The CPU time comes from running Dijkstra's algorithm once per node at stage 1. At stage 2 we insert edges in groups of $O(m/n)$, running Dijkstra's algorithm after each insertion, until we have inserted $|pending| - n/2 = O(m)$ edges overall. This accounts for other n times we run Dijkstra's algorithm. Hence the $O(nm \log m)$ complexity.

This algorithm reduces both CPU time and memory requirements, but the amount of distance evaluations is very high ($O(nm) \geq O(n^2)$).

```
t-Spanner2 (Stretch t, Vertices 𝕌)

t-Spanner ← t-spanner edge structure // initially has the first MST
pending ← ∅ // global pending egde list
H1 ← |E_{t-Spanner1}(n, d, t)| / n

Stage 1: generating t-Spanner and pending
for u ∈ 𝕌
   if |pending| > 1.2 · |t-Spanner| // using H2
      t-Spanner ← t-Spanner ∪ MST // built over the edges not yet inserted
   distances ← Dijkstra(t-Spanner, u) // distances(v) = d_{t-Spanner}(u, v)
   for v ∈ 𝕌
      if distance(v) ≤ t · d(u, v) then pending ← pending − {(u, v)}
      else pending ← pending ∪ {(u, v)}
   if |pending(u)| ≤ H1/2
      t-Spanner ← t-Spanner ∪ pending(u), pending ← pending − pending(u)

Stage 2: Reducing pending
while |pending| > n/2
   pendingNodes ← nodes sorted in decreasing number of pending edges
   for u ∈ pendingNodes
      if more than n edges have been inserted break // special case 1
      if |pending(u)| < H1/4 // special case 2
         t-Spanner ← t-Spanner ∪ pending(u), pending ← pending − pending(u)
      else
         distances ← Dijkstra(t-Spanner, u)
         for v ∈ pending(u)
            if distances(v) ≤ t · d(u, v) then pending ← pending − {(u, v)}
         smallest ← H1/4 smallest edges ∈ pending(u)
         t-Spanner ← t-Spanner ∪ smallest, pending ← pending − smallest

Stage 3: t-Spanner ← t-Spanner ∪ pending
```

Fig. 4. Massive edges insertion algorithm (t-Spanner 2), $pending(u)$ denotes $\{e \in pending,\ \exists v,\ e = (u, v)\}$.

6 Incremental Node Insertion Algorithm

This version reduces the amount of distance evaluations to just $n(n-1)/2$, while preserving the amortization update cost idea.

This algorithm, unlike the previous ones, makes a local analysis of nodes and edges. We insert the nodes one by one, not the edges. The invariant is that for nodes $1 \ldots i-1$ we have a well formed t-spanner, and

we want to insert the i-th node to the growing t-spanner. Since the insertion process only locally analyzes the edge set, the resulting t-spanner is suboptimal.

For each new node i, the algorithm makes two operations: the first is to connect the node to the growing t-spanner using the cheapest edge (towards a node $< i$); the second one is to verify that the distance estimations satisfy the t-condition, adding some edges to node i until the invariant is restored. We repeat this process until we insert the whole node set.

We also use the H_1 heuristic, with the difference that we recompute H_1 at every iteration (since the t-spanner size changes). We fixed that the number of edges to insert at a time should be $\delta = H_1/(5 \cdot i)$.

For the distance verification we use an incremental Dijkstra's algorithm with limited propagation, that is, the first time, Dijkstra's algorithm takes an array with precomputed distances initialized at $t \cdot d(u_i, u_j) + \varepsilon$, with $\varepsilon > 0$, $j \in [1, i-1]$. For the next iterations, Dijkstra's algorithm reuses the previously computed array. This is used to limit the propagation of the recomputations: a node adds its edges to the process only if its estimation improves.

Figure 5 depicts the incremental node insertion algorithm. This algorithm takes $O(n^2)$ distance evaluations, $O(nm \log m)$ CPU time, and $O(n+m) = O(m)$ memory. The CPU time comes from the fact that every node runs Dijkstra's algorithm $n/\delta = O(1)$ times.

```
t-Spanner3 (Stretch t, Vertices 𝕌)

t-Spanner ← t-spanner edge structure // initially ∅

for i ∈ [1, n]
   δ ← |E_{t-Spanner1}(i, d, t)| / (i · 5) // incremental H1
   k ← argmin_{j∈[1,i−1]}{d(node_i, node_j)}
   t-Spanner ← t-Spanner ∪ {(node_i, node_k)} // inserting the cheapest edge
   distances ← {(node_j, t · d(node_i, node_j) + ε), j ∈ [1, i − 1]} // defining the propagation limit
   while node_i has wrongly t-estimated edges
      distances ← Dijkstra(t-Spanner, u, distances)// incremental Dijkstra
      pending_i ← {(node_i, node_j), j < i, distance(node_j) > t · d(node_i, node_j)}
      smallest ← δ cheapest edges in pending_i
      t-Spanner ← t-Spanner ∪ smallest
```

Fig. 5. Incremental node insertion algorithm (t-Spanner 3).

7 Recursive Algorithm

The incremental algorithm is a good approach to construct t-spanners, but it does not consider spatial proximity (or remoteness) among the objects. A way to solve this is that the set in which the t-spanner is incrementally built is made up of near objects. Following this principle, we present a solution that recursively divides the object set into two compact subsets, builds sub-t-spanners in the subsets, and then merges them.

For the initial set division we take two far away objects, p_1 and p_2, that we call *representatives*, and then generate two subsets: objects nearer to p_1 and nearer to p_2. Figure 6 (left) shows the concept graphically. For the recursive divisions we reuse the representative as one of the two objects, and the element farthest to it as the other. The recursion finishes when we have less than 3 objects.

The merge step also takes into account the spatial proximity among the objects. When we merge the sub-t-spanners, we have two node subsets V_1 and V_2, where $|V_1| \geq |V_2|$ (otherwise we swap the subsets). Then, in the sub-t-spanner represented by p_2 ($stsp_2$), we choose the object closest to p_1 (u), and insert it into

Fig. 6. On the left, we select p_1 and p_2, and then divide the set. On the right, the merge step takes the objects according to their distances towards p_1.

the sub-t-spanner represented by p_1 ($stsp_1$) verifying that all the distances towards V_1 are well t-estimated. Note that this is equivalent to consider that we use the incremental algorithm, where we insert u into the growing t-spanner $stsp_1$. We continue with the second closest and repeat the procedure until all the $stsp_2$ nodes are inserted into $stsp_1$. Figure 6 (right) illustrates. Note that the edges already present in $stsp_2$ are conserved.

This algorithm also uses an incremental Dijkstra's algorithm with limited propagation, but this time we are only interested in limiting the propagation towards $stsp_1$ nodes (because we know that towards $stsp_2$ we already satisfy the t-condition). Hence, Dijkstra's algorithm takes an array with precomputed distances initialized at $t \cdot d(u_i, u_j) + \varepsilon$ for $(u_i, u_j) \in V_2 \times V_1$, and ∞ for $(u_i, u_j) \in V_2 \times V_2$, where ε is a small positive constant. For the next iterations, Dijkstra's algorithm reuses the previously computed array.

Figure 7 depicts the recursive algorithm and the auxiliary functions used to build and merge sub-t-spanners. This algorithm takes $O(n^2)$ distance evaluations, $O(nm \log m)$ CPU time, and $O(n+m) = O(m)$ memory. The cost of dividing the sets does not affect that of the underlying incremental construction.

8 Experimental Results

We have tested our algorithms on a synthetic set of random points in a k-dimensional space. However, we have not used the fact that the space has coordinates, treating the points as abstract objects in an unknown metric space. This choice permits us to show how the algorithms perform as a function of the dimension. In the final version we will include experiments on other metric spaces such as strings using edit distance.

We are interested in measuring the CPU time needed and the amount of edges generated by each algorithm. Figures 8 and 9 show a comparison among the four algorithms. As it can be seen, the optimized basic algorithm is impractically costly, but it produces the best t-spanners. Depending on the dimension, the next best-quality spanners are produced by the recursive algorithm (low dimensions) or by the massive edge insertion algorithm (high dimensions). Note, however, that the differences in spanner quality become less important for higher dimensions.

It is interesting to notice that, even in dimension 28 and for $t = 1.8$, the number of edges in the spanners is still less than 10% of the complete graph. Fitting the experimental results of t-spanner1 we obtain the following empirical model, which we use for the H_1 heuristic:

$$|E_{t-\text{Spanner }1}(n, dim, t)| = 0.036\ dim^{\frac{4.2}{t^{2.1}}+3.7\frac{\ln t}{\ln dim}} n^{1.17} \tag{2}$$

We show now more massive tests excluding t-spanner1, which is too costly. Figures 10, 11 and 12 give the results. The conclusions are quite similar, except that for large n it is not always clear which is faster between the incremental and the recursive algorithms.

Table 2 shows our least squares fittings on the data. As it can be seen, for $dim \in [4, 28]$ and $t \in [1.3, 2.0]$, all our algorithms except the basic optimized algorithm take from $O(n^{2.18})$ to $O(n^{2.32})$ time and produce t-spanners with $O(n^{1.12})$ to $O(n^{1.14})$ edges. The basic optimized algorithm, on the other hand, is $O(n^{3.19})$ time,

```
t-Spanner4 (Stretch t, Vertices 𝕌)

t-Spanner ← t-spanner edge structure // initially ∅
(p1, p2) ← two distant objects
(V1, V2) ← 𝕌 divided according to distances towards (p1, p2)
stsp1 ← makeSubtSpanner(p1,V1), stsp2 ← makeSubtSpanner(p2,V2)
t-Spanner ← mergeSubtSpanner(stsp1, stsp2)

makeSubtSpanner(representative p, Vertices V)
if |V| = 1 return t-spanner (nodes = {p}, edges = ∅)
else if |V| = 2 return t-spanner (nodes = V = {v1, v2}, edges = {(v1, v2)})
else
   p_remote ← argmax_{v∈V} {d(p, v)}
   (V, V_remote) ← V divided according to distances towards (p, p_remote)
   stsp_p ← makeSubtSpanner(p,V), stsp_remote ← makeSubtSpanner(p_remote,V_remote)
   return mergeSubtSpanner(stsp_p, stsp_remote)

mergeSubtSpanner (t-Spanner stsp1, t-Spanner stsp2)
if |nodes(stsp1)| ≤ |nodes(stsp2)| stsp1 ⇔ stsp2
nodes ← nodes(stsp1) ∪ nodes(stsp2)
edges ← edges(stsp1) ∪ edges(stsp2)
δ ← |E_{t−Spanner1}(|nodes|, d, t)| / (i · 5) // incremental H1
p1 ← representative(stsp1)
for u ∈nodes(stsp2) in increasing order of d(u, p1)
   for v ∈ nodes(stsp1) do distances(v) ← t · d(u, v) + ε // defining the propagation limit towards stsp1
   for v ∈nodes(stsp2) do distances(v) ← ∞
   while u has wrongly t-estimated edges towards stsp1
      distances ← Dijkstra(edges, u, distances)// incremental Dijkstra
      pending_u ← {(u, v), v ∈ stsp1, distance(v) > t · d(u, v)}
      smallest ← δ cheapest edges ∈ pending_u
      edges ← edges ∪ smallest
return t-Spanner (nodes = nodes, edges = edges)
```

Fig. 7. Recursive algorithm (t-Spanner 4).

and it corresponds to the currently most widely used technique. This shows that our algorithms represent in practice a giant improvement over the current state of the art.

9 Conclusions

We have presented several algorithms for t-spanner construction when the underlying graph is the complete graph representing distances in a metric space. This is motivated by our recent research on searching metric spaces and shows that t-spanners are well suited as data structures for this problem. For this sake, we need practical construction algorithms for $1 < t \leq 2$. To the best of our knowledge, no existing technique works well under this scenario (complete graph, metric distances, small t, practical construction time). However, our algorithms are also well suited to general graphs.

Our focus has been on practical algorithms. We have shown that it is possible to build good quality t-spanners in reasonable time. For example, where the metric space turns out to be the Euclidean unitary cube in d dimensions, for $4 \leq d \leq 28$, and $1.4 \leq t \leq 2.0$ (which is quite stringent compared to the literature), we have built t-spanners with $O(n^{1.13})$ edges in $O(n^{2.24})$ time. Note that just computing the minimum spanning

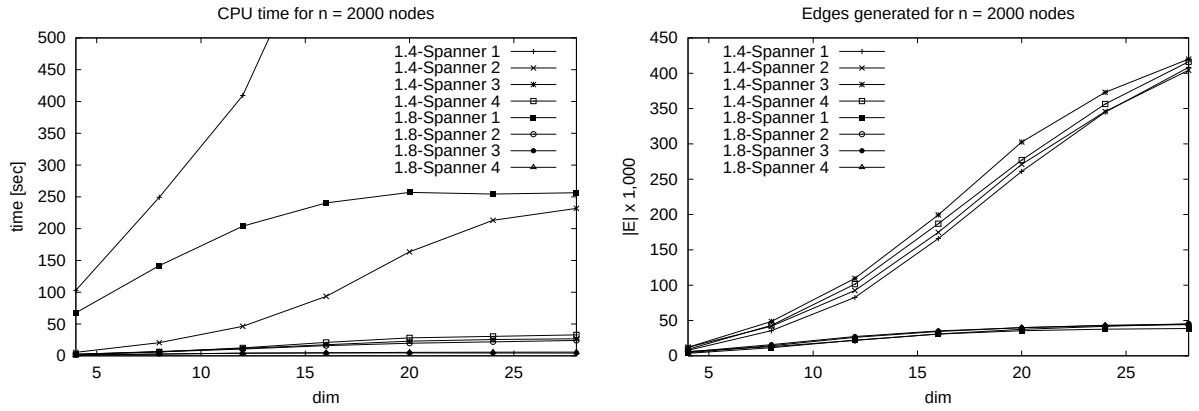

Fig. 8. On the left, construction time versus dimension. On the right, number of edges generated versus dimension. The number of elements is $n = 1000$. t-Spanner1 reaches 2,500 seconds in $dim = 28$.

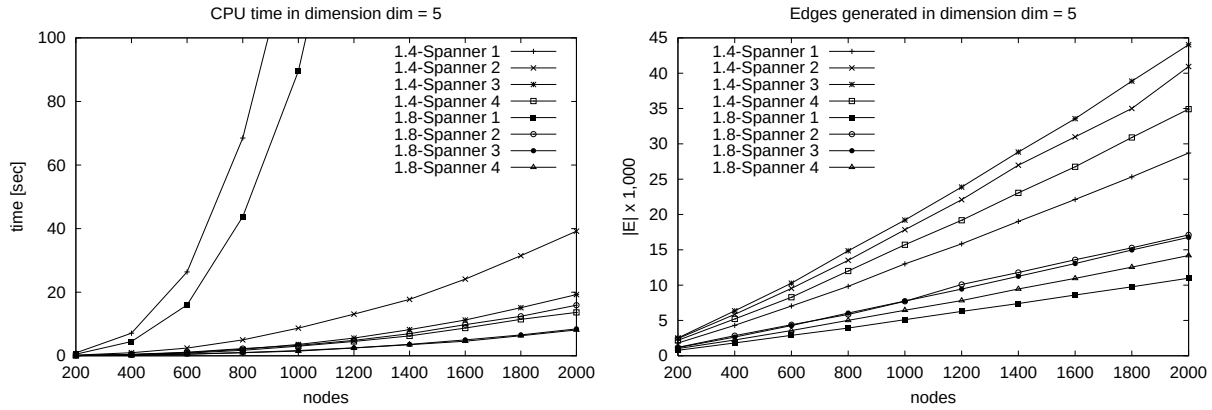

Fig. 9. On the left, construction time versus number of nodes n. On the right, number of edges generated versus n. The dimension is $dim = 5$. t-Spanner1 reaches 1,250 seconds for $n = 2000$ and $t = 1.4$.

tree requires $O(n^2)$ time. Compared to the existing algorithms, our contribution represents in practice a giant improvement over the current state of the art.

It is possible to add and remove elements from the spanner in reasonable time while preserving its quality. The incremental algorithm permits adding new elements. Remotion of a node can be arranged by adding a clique among its neighbors and periodically reconstructing the spanner with the recursive algorithm.

Future work involves using t-spanners where t depends on the actual distance between the nodes. Basically, we are more interested in approximating well short rather than long distances. Another trend is on probabilistic t-spanners, where distances are well t-estimated with high probability.

References

1. I. Althöfer, G. Das, D. Dobkin, and D. Joseph. Generating sparse spanners for weighted graphs. In *Proc. 2nd Scandinavian Workshop on Algorithm Theory (SWAT'90)*, LNCS 447, pages 26–37, 1990.
2. I. Althöfer, G. Das, D. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete Computational Geometry*, 9:81–100, 1993.

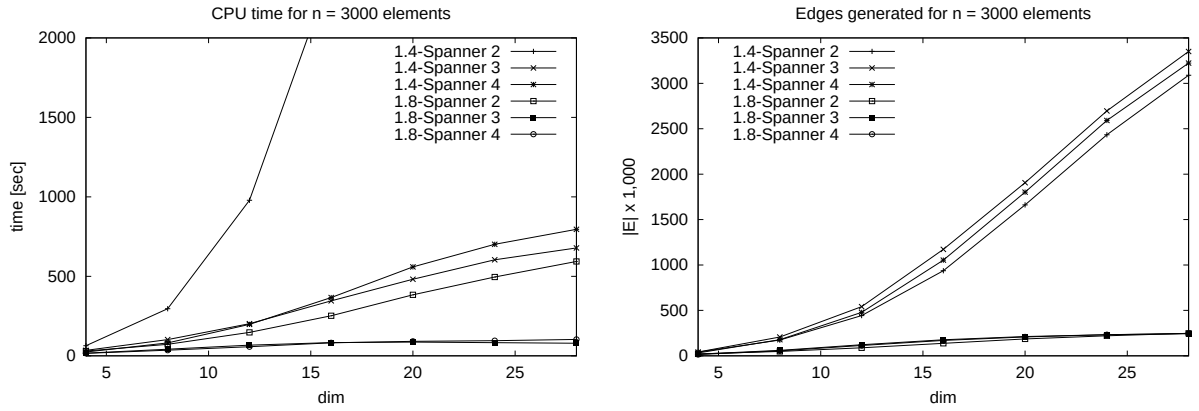

Fig. 10. On the left, construction time versus dimension. On the right, edges generated versus dimension. We used $n = 3000$ nodes. t-Spanner2 reaches 14,000 seconds for $dim = 28$.

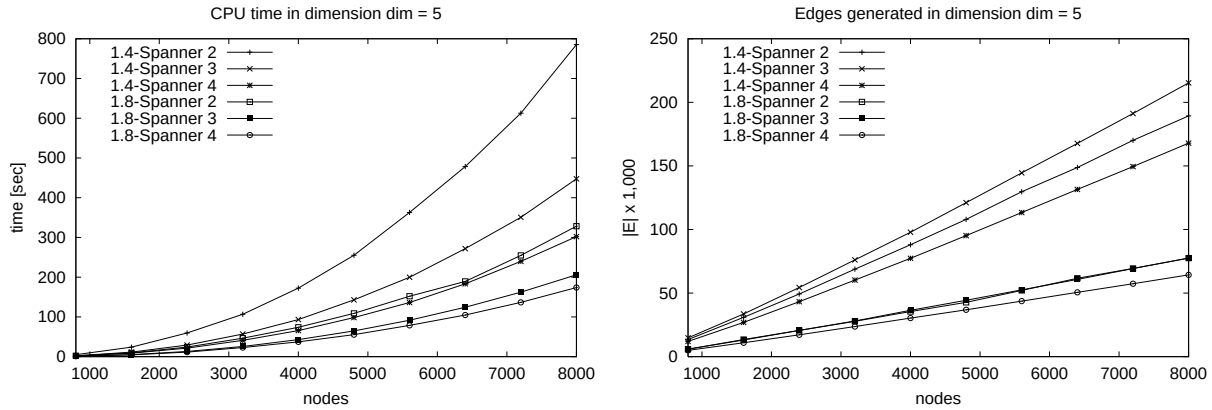

Fig. 11. On the left, construction time versus n. On the right, edges generated versus n. We used $dim = 5$.

3. Y. Bartal. On approximating arbitrary metrics by tree metrics. In *Proc. 30th Symposium on the Theory of Computing (STOC'98)*, pages 161–168, 1998.
4. M. Charikar, C. Chekuri, A. Goel, S. Guha, and S. Plotkin. Approximating a finite metric by a small number of tree metrics. In *Proc. 39th Symp. on Foundations of Computer Science (FOCS'98)*, pages 379–388, 1998.
5. E. Chávez, G. Navarro, R. Baeza-Yates, and J.L. Marroquin. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.
6. E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM J. on Computing*, 28:210–236, 1998.
7. D. Eppstein. Spanning trees and spanners. In *Handbook of Computational Geometry*, pages 425–461. Elsevier, 1999.
8. J. Gudmundsson, C. Levcopoulos, and G. Narasimhan. Improved greedy algorithms for constructing sparse geometric spanners. In *Proc. 7th Scandinavian Workshop on Algorithm Theory (SWAT 2000)*, LNCS v. 1851, pages 314–327, 2000.
9. J.M. Keil. Approximating the complete Euclidean graph. In *Proc. 1st Scandinavian Workshop in Algorithm Theory (SWAT'88)*, LNCS 318, pages 208–213, 1988.
10. W. Liang and R. Brent. Constructing the spanners of graphs in parallel. Technical Report TR-CS-96-01, Dept. of CS and CS Lab, The Australian National University, January 1996.

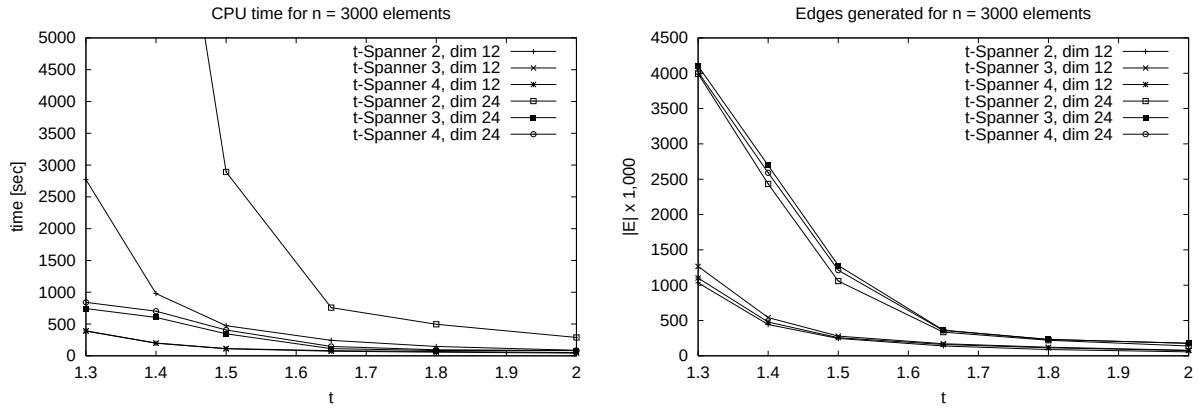

Fig. 12. On the left, construction time versus t. On the right, edges generated versus t. We used $n = 3000$ elements. t-Spanner2 reaches 18,500 seconds for $dim = 24$ and $t = 1.3$.

	Basic optimized	Massive edge insertion	Incremental	Recursive
CPU time	$O(n^{3.19})$	$O(n^{2.18})$	$O(n^{2.32})$	$O(n^{2.24})$
Edges	$O(n^{1.16})$	$O(n^{1.12})$	$O(n^{1.14})$	$O(n^{1.13})$

Table 2. Empirical complexities of our algorithms. For simplicity we are hiding the dependence with the dimension and t.

11. G. Navarro, R. Paredes, and E. Chávez. t-Spanners as a data structure for metric space searching. Submitted for publication, 2002.
12. D. Peleg and A. Schaffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.
13. D. Peleg and J. Ullman. An optimal synchronizer for the hypercube. *SIAM J. on Computing*, 18:740–747, 1989.
14. J. Ruppert and R. Seidel. Approximating the d-dimensional complete Euclidean graph. In *3rd Canadian Conference on Computational Geometry*, pages 207–210, 1991.
15. E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Patt. Recog. Lett.*, 4:145–157, 1986.
16. Mark Allen Weiss. *Data Structures and Algorithm Analysis*. Addison-Wesley, 2nd edition, 1995.