

# $t$ -Spanners for Metric Space Searching<sup>★</sup>

Gonzalo Navarro<sup>a</sup>, Rodrigo Paredes<sup>a,\*</sup>, Edgar Chávez<sup>b</sup>

<sup>a</sup> *Center for Web Research, Dept. of Computer Science, University of Chile.  
Blanco Encalada 2120, Santiago, Chile.*

<sup>b</sup> *Escuela de Ciencias Físico-Matemáticas, Univ. Michoacana, Morelia, México.*

---

## Abstract

The problem of *Proximity Searching in Metric Spaces* consists in finding the elements of a set which are close to a given query under some similarity criterion. In this paper we present a new methodology to solve this problem, which uses a  $t$ -spanner  $G'(V, E)$  as the representation of the metric database. A  $t$ -spanner is a subgraph  $G'(V, E)$  of a graph  $G(V, A)$ , such that  $E \subseteq A$  and  $G'$  approximates the shortest path costs over  $G$  within a precision factor  $t$ .

Our key idea is to regard the  $t$ -spanner as an approximation to the complete graph of distances among the objects, and to use it as a compact device to simulate the large matrix of distances required by successful search algorithms such as AESA. The  $t$ -spanner properties imply that we can use shortest paths over  $G'$  to estimate any distance with bounded-error factor  $t$ .

For this sake, several  $t$ -spanner construction, updating, and search algorithms are proposed and experimentally evaluated. We show that our technique is competitive against current approaches. For example, in a metric space of documents our search time is only 9% over AESA, yet we need just 4% of its space requirement. Similar results are obtained in other metric spaces.

Finally, we conjecture that the essential metric space property to obtain good  $t$ -spanner performance is the existence of clusters of elements, and enough empirical evidence is given to support this claim. This property holds in most real-world metric spaces, so we expect that  $t$ -spanners will display good behavior in most practical applications. Furthermore, we show that  $t$ -spanners have a great potential for improvements.

*Key words:* Indexing methods, Similarity-based operations, Proximity searching, Graphs

---

## 1 Introduction

*Proximity searching* is the problem of, given a data set and a similarity criterion, finding the elements of the set that are close to a given query. This problem has a vast number of applications. Some examples are:

- *Non-traditional databases.* New so-called “multimedia” data types such as images, fingerprints, audio and video cannot be meaningfully queried in the classical sense. In multimedia applications, all the queries ask for objects *similar* to a given one, whereas comparison for exact equality is very rare. Some example applications are image, audio or video databases, face recognition, fingerprint matching, voice recognition, medical databases, and so on.
- *Text retrieval.* Huge text databases with low quality control are emerging (being the Web the most prominent example), and typing, spelling or OCR (optical character recognition) errors are commonplace in both the text and the query. Documents which contain a misspelled word are no longer retrievable by a correctly written query. Thus many text search engines aim to find documents containing close variants of the query words. There exist several models of similarity among words (variants of the “edit distance” [32]) which capture very well those kind of errors. Another related application is spelling checkers, where we look for close variants of a misspelled word in a dictionary.
- *Information retrieval.* Although not considered as a multimedia data type, unstructured text retrieval poses problems similar to multimedia retrieval. This is because textual documents are in general not structured to easily provide the desired information. Although text documents may be searched for strings that are present or not, in many cases it is more useful to search them for semantic concepts of interest. The problem is basically solved by retrieving documents similar to a given query [5], where the query can be a small set of words or even another document. Some similarity approaches are based on mapping a document to a vector of real values, so that each dimension is a vocabulary word and the relevance of the word to the document (computed using some formula) is the coordinate of the document along that dimension. Similarity functions are then defined on that space. Notice, however, that the dimensionality of the space is very high (thousands of dimensions).

---

\* This work has been supported in part by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile; MECESUP Project UCH0109, Chile; and CONACyT, Mexico.

\* Corresponding author.

*Email addresses:* `gnavarro@dcc.uchile.cl` (Gonzalo Navarro),  
`raparede@dcc.uchile.cl` (Rodrigo Paredes), `elchavez@umich.mx` (Edgar Chávez).

- *Computational biology.* DNA and protein sequences are the basic objects of study in molecular biology. They can be modeled as texts, and in this case many biological quests translate into finding local or global similarities between sequences, in order to detect homologous regions that permit predicting functionality, structure or evolutionary distance. An exact match is unlikely to occur because of measurement errors, minor differences in genetic streams with similar functionality, and evolution. The measure of similarity used is related to the probability of mutations such as reversals of pieces of the sequences and other rearrangements (global similarity), or variants of edit distance (local similarity).
- There are many other applications, such as machine learning and classification, where a new element must be classified according to its closest existing element; image quantization and compression, where only some vectors can be represented and those that cannot must be coded as their closest representable point; function prediction, where we want to search for the most similar behavior of a function in the past so as to predict its probable future behavior; and so on.

All those applications have some common characteristics, captured under the *metric space model* [16]. There is a universe  $\mathbb{X}$  of *objects*, and a nonnegative *distance function*  $d : \mathbb{X} \times \mathbb{X} \rightarrow \mathbb{R}^+ \cup \{0\}$  defined among them. This distance satisfies the three axioms that make  $(\mathbb{X}, d)$  a *metric space*:

$$\begin{aligned}
 d(x, y) = 0 & \iff x = y && \text{(strict positiveness)} \\
 d(x, y) &= d(y, x) && \text{(symmetry)} \\
 d(x, z) &\leq d(x, y) + d(y, z) && \text{(triangle inequality)}
 \end{aligned}$$

These properties hold for many reasonable similarity functions. The smaller the distance between two objects, the more “similar” they are. We have a finite *database*  $\mathbb{U} \subseteq \mathbb{X}$ , which is a subset of the universe of objects and can be preprocessed to build an *index*. Later, given a new object from the universe, a *query*  $q \in \mathbb{X}$ , we must retrieve similar elements found in the database. There are two typical queries of this kind:

**Range query**  $(q, r)$ : Retrieve all elements which are within distance  $r$  to  $q$ .

That is,  $(q, r) = \{x \in \mathbb{U}, d(x, q) \leq r\}$ .

**$k$ -Nearest neighbor query**  $NN_k(q)$ : Retrieve the  $k$  elements from  $\mathbb{U}$  closest to  $q$ . That is,  $NN_k(q)$  such that  $\forall x \in NN_k(q), y \in \mathbb{U} - NN_k(q), d(q, x) \leq d(q, y)$ , and  $|NN_k(q)| = k$ .

The distance is assumed to be expensive to compute (think, for instance, in comparing two fingerprints). Hence, it is customary to define the complexity of the search as the number of distance evaluations performed, disregarding other components such as CPU time for side computations, and even I/O time. Given a database of  $n = |\mathbb{U}|$  objects, queries can be trivially answered

by performing  $n$  distance evaluations. The goal is to structure the database so as to compute much fewer distances.

A particular case of this problem arises when the space is  $\mathbb{R}^D$ , for which there are effective methods such as kd-trees, R-trees, and X-trees [23,7]. However, for high dimensions those structures cease to work well. In this paper we focus in general metric spaces, although these solutions are also well suited to  $D$ -dimensional spaces.

It is interesting to notice that the concept of “dimensionality” can be translated into metric spaces as well, where it is called *intrinsic dimensionality*. Although there is not an accepted criterion to measure intrinsic dimensionality, in general terms, it is agreed that a metric space is high dimensional—that is, it has high intrinsic dimensionality—when its histogram of distances is concentrated. A high intrinsic dimension degrades the performance of any similarity search algorithm [16].

By far, the most successful technique for searching metric spaces ever proposed is AESA [40,16]. Its main problem, however, is that it requires precomputing and storing a matrix of all the  $n(n-1)/2$  distances among the objects of  $\mathbb{U}$ . This huge space requirement makes it unsuitable for most applications.

Let us now switch to graph theory and the  $t$ -spanner concept. Let  $G(V, A)$  be a connected graph 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 costs of the traversed edges. Let us call  $d_G(u, v)$  this shortest path cost (or minimum sum). Shortest paths can be computed using Floyd’s algorithm [22] or applying  $|V|$  iterations of Dijkstra’s algorithm [20] taking each vertex as the origin node [19,41]. A  $t$ -spanner is a subgraph  $G'(V, E)$ , with  $E \subseteq A$ , which permits us to compute path costs with *stretch*  $t$ , that is, ensuring that for any  $u, v \in V$ ,  $d_{G'}(u, v) \leq t \cdot d_G(u, v)$  [34,33,21]. We call the latter the  $t$ -condition. The  $t$ -spanner problem has applications in distributed systems, communication networks, architecture of parallel machines, motion planning, robotics, computational geometry, and others.

Our main idea is to combine both concepts, so as to use the  $t$ -spanner as a bounded-error approximation to the full AESA distance matrix, in order to obtain a competitive space–time trade-off for metric space searching.

Hence, our interest in this paper is on the design and evaluation of  $t$ -spanner algorithms that work well in metric space contexts. To achieve this goal, several algorithms to build, maintain, and search  $t$ -spanners are proposed. All these algorithms are experimentally evaluated, to show that our approach provides an excellent low-cost approximation to the performance of AESA. As an example, we apply the idea to information retrieval, with document databases using the cosine distance [5]. If we use a 2-spanner, we need only

3.99% of the memory required by AESA for the index. To retrieve the most similar document, we need only 9% distance evaluations over AESA, and 8% to retrieve the 10 most similar documents.

This paper is organized as follows. In Section 2 we cover related work both in metric spaces and in  $t$ -spanners. In Section 3 we show how to use  $t$ -spanners for metric space searching. In Section 4 we present our  $t$ -spanner construction algorithms. Experimental results are shown in Section 5. In Section 6 we discuss  $t$ -spanner updating algorithms. Finally, in Section 7 we draw our conclusions and future work directions. Early versions of this work appeared in [31,30].

## 2 Related Work

### 2.1 Searching in Metric Spaces

There are several methods to preprocess a metric database in order to reduce the number of distance evaluations at search time. All of them work by discarding elements using the triangle inequality. We discuss here some approaches that are relevant to our work. See [16] for a more complete survey.

#### 2.1.1 Pivot-based Algorithms

We will focus on a particular class of algorithms called *pivot-based*. These algorithms select a set of pivots  $\{p_1 \dots p_k\} \subseteq \mathbb{U}$  and store a table of  $kn$  distances  $d(p_i, u)$ ,  $i \in \{1 \dots k\}, \forall u \in \mathbb{U}$ . To solve a range query  $(q, r)$ , pivot-based algorithms measure  $d(q, p_1)$  and use the fact that, because of the triangle inequality,  $d(q, u) \geq |d(q, p_1) - d(u, p_1)|$ , so they can discard every  $u \in \mathbb{U}$  such that

$$|d(q, p_1) - d(u, p_1)| > r, \quad (1)$$

since this implies  $d(q, u) > r$ . Once they are done with  $p_1$ , they try to discard elements from the remaining set using  $p_2$ , and so on, until they use all the  $k$  pivots. The elements  $u$  that still cannot be discarded at this point are directly compared against  $q$ .

The  $k$  distance evaluations computed between  $q$  and the pivots are known as the *internal complexity* of the algorithm. If there is a fixed number of pivots, this complexity has a fixed value. On the other hand, the distance evaluations used to compare the query against the objects not discarded by the pivots are known as the *external complexity* of the algorithm. Hence, the total complexity of a pivot-based search algorithm is the sum of the internal

and external complexities. Since the internal complexity increases and the external complexity decreases with  $k$ , there is an optimal  $k^*$  that minimizes the total complexity. However, in practice  $k^*$  is so large that one cannot store all the  $k^*n$  distances, so the index uses as many pivots as memory permits.

There are many pivot-based algorithms. Among them we can find structures for discrete or continuous distance functions. In the discrete case we have: *Burkhard-Keller Tree* (BKT) [10], *Fixed Queries Tree* (FQT) [4], *Fixed-Height FQT* (FHQT) [4,3], and *Fixed Queries Array* (FQA) [14]. In the continuous case we have: *Vantage-Point Tree* (VPT) [42,17,39], *Multi-Vantage-Point Tree* (MVPT) [9,8], *Excluded Middle Vantage Point Forest* (VPF) [43], *Approximating Eliminating Search Algorithm* (AESA) [40], and *Linear AESA* (LAESA) [29]. For a comprehensive description of these algorithms see [16].

### 2.1.2 Approximating Eliminating Search Algorithm (AESA)

In AESA the idea of pivots is taken to the extreme  $k = n$ , that is, every element is a potential pivot and hence we need a matrix with all the  $\frac{n(n-1)}{2}$  precomputed distances. Since we are free to choose any pivot, the pivot to use next is chosen from the elements not yet discarded. Additionally, as it is well known that pivots closer to the query are much more effective, the pivot candidates  $u$  are ranked according to the sum of their current lower-bound distance estimations to  $q$ . That is, if we have used pivots  $\{p_1 \dots p_i\}$ , we choose pivot  $p_{i+1}$  as the element  $u$  minimizing

$$SumLB(u) = \sum_{j=1}^i |d(q, p_j) - d(u, p_j)| \quad (2)$$

AESA works as follows. It starts with a set of candidate objects  $\mathcal{C}$ , which is initially  $\mathbb{U}$ , and sets  $SumLB(u) = 0$  for all  $u \in \mathbb{U}$ . Then, it chooses an object  $p \in \mathcal{C}$  minimizing  $SumLB$  (Eq. (2)) and removes it from  $\mathcal{C}$ . Note that the first object  $p = p_1$  is chosen at random. AESA measures  $d_{qp} \leftarrow d(q, p)$  and reports  $p$  if  $d_{qp} \leq r$ . By Eq. (1), it removes from  $\mathcal{C}$  all the objects  $u$  that satisfy  $d(u, p) \notin [d_{qp} - r, d_{qp} + r]$ . Recall that  $d(u, p)$  is obtained from a precomputed full distance matrix  $\mathbb{U} \times \mathbb{U}$ . For non-discarded objects, it updates  $sumLB$  according to Eq (2). These steps are repeated until  $\mathcal{C} = \emptyset$ . Fig. 1 depicts the algorithm.

AESA is, by far, the most efficient existing search algorithm. As a matter of fact, it has been experimentally shown to have almost constant search cost. Nevertheless, the constant hides an exponential dependence on the dimensionality of the metric space. AESA's main problem is that storing  $O(n^2)$  distances is impractical for most applications. This has restricted an excellent algorithm to the few applications where  $n$  is very small. Our main goal in this paper is

---

```

AESA (Query  $q$ , Radius  $r$ , matrix  $\mathbb{M}$ )
  //  $\mathbb{M}_{u,p} = d(u,p)$ , precomputed
  1.  $\mathcal{C} \leftarrow \mathbb{U}$ 
  2. For each  $p \in \mathcal{C}$  Do  $SumLB(p) \leftarrow 0$ 
  3. While  $\mathcal{C} \neq \emptyset$  Do
  4.    $p \leftarrow \operatorname{argmin}_{c \in \mathcal{C}} SumLB(c)$ ,  $\mathcal{C} \leftarrow \mathcal{C} - \{p\}$ 
  5.    $d_{qp} \leftarrow d(q,p)$ , If  $d_{qp} \leq r$  Then Report  $p$ 
  6.   For each  $u \in \mathcal{C}$  Do
  7.     If  $\mathbb{M}_{u,p} \notin [d_{qp} - r, d_{qp} + r]$  Then  $\mathcal{C} \leftarrow \mathcal{C} - \{u\}$ 
  8.     Else  $SumLB(u) \leftarrow SumLB(u) + |d_{qp} - \mathbb{M}_{u,p}|$ 

```

---

Fig. 1. Algorithm AESA.  $\mathbb{M}$  is the full distance matrix  $\mathbb{U} \times \mathbb{U}$ , so  $\mathbb{M}_{u,p}$  is the distance between  $u$  and  $p$ .

to overcome this weakness.

### 2.1.3 Previous Graph-Based Approaches

We have found only one previous metric space index based on graphs [37]. They use a graph whose nodes are the objects of the metric space and whose edges are an *arbitrary* collection of distances among the objects. They compute two  $n \times n$  matrices with upper and lower bounds to the real distances, in  $O(n^3)$  time. Queries are solved using both matrices.

The greatest deficiency of [37] is that the selected distances are arbitrary and do not give any guarantee on the quality of their approximation to the real distances. In fact, the index only has good behavior when distances follow a uniform distribution, which does not occur in practice. Even in  $\mathbb{R}$ , an extremely easy-to-handle metric space, distances have a triangular distribution, whereas in general metric spaces the distance distribution is usually more concentrated, far from uniform.

In our work, instead, edges are carefully chosen so as to provide an approximation guarantee to the real distance value. This yields much better results.

## 2.2 $t$ -Spanner Construction Algorithms

It was shown [33] that, given a graph  $G$  with unitary weight edges and parameters  $t$  and  $m$ , the problem of determining whether a  $t$ -spanner of at most  $m$  edges exists is NP-complete. Hence, there is no hope for efficient construction of minimal  $t$ -spanners. Furthermore, as far as we know, only in the case  $t = 2$  and graphs with unitary weight edges there exist polynomial-time algorithms that guarantee an approximation bound in the number of edges (or in the

weight) of the resulting  $t$ -spanner [27] (the bound is  $\log \frac{|E|}{|V|}$ ).

If we do not force any guarantee on the number of edges of the resulting  $t$ -spanner, a simple  $O(mn^2)$  time greedy algorithm exists (see Section 4.1), where  $n = |V|$  and  $m = |E|$  refer to the resulting  $t$ -spanner. It was shown [1,2] that these techniques produce  $t$ -spanners with  $n^{1+O(\frac{1}{t-1})}$  edges on general graphs of  $n$  nodes.

More sophisticated algorithms have been proposed by Cohen in [18], producing  $t$ -spanners with guaranteed  $O(n^{1+(2+\varepsilon)(1+\log_n m)/t})$  edges in worst case time  $O(mn^{(2+\varepsilon)(1+\log_n m)/t})$ , where in this case  $m$  refers to the original graph. In our metric space application  $m = \Theta(n^2)$ , which translates into worst case time  $O(n^{2+6/t})$  and  $O(n^{1+6/t})$  edges for Cohen’s techniques. Additionally, the algorithms in [18] work for  $t \in [2, \log n]$ , which, as shown in Section 3, is unsuitable for our application: We need  $1 < t \leq 2$  in most cases to obtain reasonable search performance. (Perhaps some of Cohen’s algorithms could be adapted to work heuristically for smaller  $t$ , but to the best of our knowledge, this has not been attempted so far.) Note that, for  $t = 2$ , Cohen’s algorithm could still be useful, but its time complexity would be rather high,  $O(n^5)$ . Other recent algorithms [38] work only for  $t = 1, 3, 5, \dots$ , which is also unsuitable for us. Parallel algorithms have been pursued in [28], but they do not translate into new sequential algorithms.

As it regards to Euclidean  $t$ -spanners, that is, the subclass of metric  $t$ -spanners where objects are points in a  $D$ -dimensional space with Euclidean distance, much better results exist [21,1,2,26,25,36], 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 [6,12]. 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  $O(\log n \log \log n)$ , far from practical for our goals.

### 3 Simulating AESA Search over a $t$ -Spanner

#### 3.1 Relation between Metric Space Searching and $t$ -Spanners

As we have said, AESA is the most successful technique for searching metric spaces. However, its huge  $\frac{n(n-1)}{2} = O(n^2)$  memory demand makes it unsuitable for most applications. Nevertheless, if we reduced the memory requirement of

AESA, we could use it in many practical scenarios. Towards this end, our main idea is to use the  $t$ -spanner as a bounded-error approximation to the full AESA distance matrix. This permits us trading space for query time, where the full AESA is just one extreme of the trade-off.

Note that, given the element set  $\mathbb{U}$ , the full AESA distance matrix can be regarded as a complete graph  $G(V = \mathbb{U}, A = \mathbb{U} \times \mathbb{U})$ , where  $d_G(u, v) = d(u, v)$  is the distance between elements  $u$  and  $v$  in the metric space. Due to the triangle inequality, for any  $u, v \in \mathbb{U}$  the shortest path between  $u$  and  $v$  is the direct edge  $(u, v)$ , and its cost is the distance  $d(u, v)$ . Thus, in order to save memory we can use a  $t$ -spanner  $G'(V = \mathbb{U}, E \subseteq A)$  of  $G$ , which permits us estimating the distance between every pair of objects within a factor  $t$ , without the need to store  $O(n^2)$  distances but only  $|E|$  edges. However, in this case we cannot apply AESA directly over the  $t$ -spanner, but we have to take into account the error introduced by stretch factor  $t$ .

### 3.2 Implementing AESA over a $t$ -Spanner

Given the  $t$ -spanner  $G'$  of  $G(\mathbb{U}, \mathbb{U} \times \mathbb{U})$ , for every  $u, v \in \mathbb{U}$  the following property is guaranteed:

$$d(u, v) \leq d_{G'}(u, v) \leq t \cdot d(u, v) . \quad (3)$$

Eq. (3) permits us adapting AESA to this approximated distance. According to the stretch factor  $t$ , to simulate AESA over a  $t$ -spanner it is enough to “extend” the upper bound of the AESA exclusion ring with the associated decrease in the discrimination power. See Fig. 2.

Let us return to the condition to discard an element  $u$  with a pivot  $p$ . The condition to be outside the ring, that is, Eq. (1), can be rewritten as

$$d(u, p) < d(q, p) - r \quad \text{or} \quad d(u, p) > d(q, p) + r . \quad (4)$$

Since we do not know the real distance  $d(u, v)$ , but only the approximated distance over the  $t$ -spanner,  $d_{G'}(p, u)$ , we can use Eqs. (3) and (4) to obtain the new discarding conditions, in Eqs. (5) and (6):

$$d_{G'}(u, p) < d(q, p) - r , \quad (5)$$

$$d_{G'}(u, p) > t \cdot (d(q, p) + r) . \quad (6)$$

The next theorem shows that either Eq. (5) or (6) gives a sufficient condition for a pivot  $p$  to discard a element  $u$ . Fig. 2(b) illustrates.

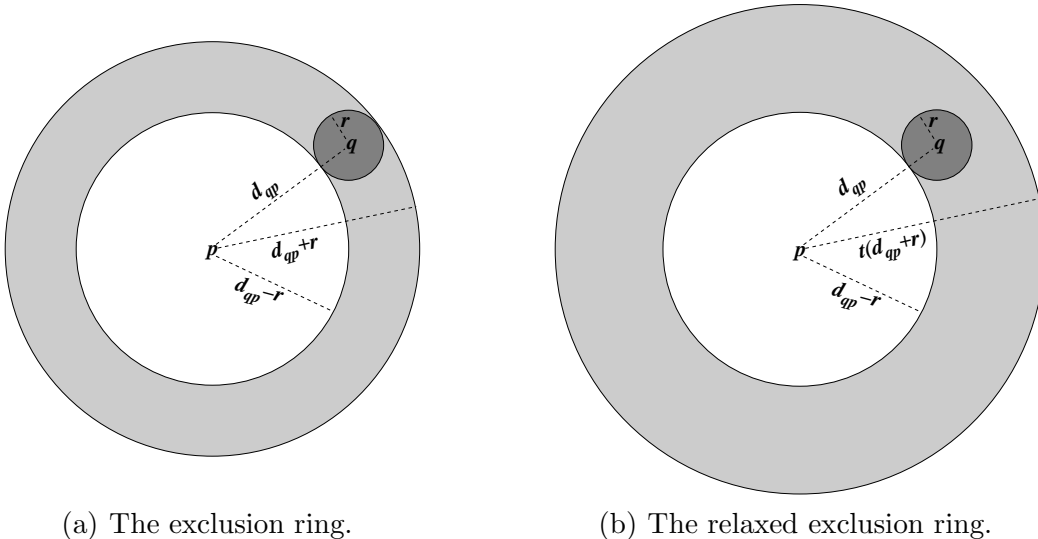


Fig. 2. In (a), the ring of elements not discarded by pivot  $p$ . In (b), the relaxed ring of elements not discarded by pivot  $p$  when using a  $t$ -spanner. We note  $d_{qp} = d(q, p)$ .

**Theorem 1 ( $t$ -discarding)** *Given a pivot  $p \in \mathbb{U}$ , a query  $q \in \mathbb{X}$  and a radius  $r \in \mathbb{R}^+$ , if  $u \in \mathbb{U}$  satisfies Eq. (5) or (6), then  $d(q, u) > r$ .*

**PROOF.** As  $d(u, p) \leq d_{G'}(u, p)$ , from Eq. (5) we obtain  $d(u, p) < d(p, q) - r$ . On the other hand, as  $d_{G'}(u, p) \leq t \cdot d(u, p)$ , from Eq. (6) we obtain that  $t \cdot d(u, p) \geq d_{G'}(u, p) > t \cdot (d(q, p) + r)$ , then  $d(u, p) > d(q, p) + r$ . Either case guarantees that  $d(q, u) > r$ .  $\square$

What we have obtained is a relaxed version of AESA, which requires less memory ( $O(|E|)$  instead of  $O(n^2)$ ) and, in exchange, discards less element per pivot. As  $t$  tends to 1, our approximation becomes better but we need more and more edges. Hence we have a space–time trade-off where full AESA is just one extreme.

Let us now consider how to choose the next pivot. Since we have only an approximation to the true distance, we cannot directly use Eq. (2). To compensate for the effect of the precision factor  $t$ , after some experimental fine-tuning, we have chosen  $\alpha_t = \frac{2/t+1}{3}$ , so as to rewrite Eq. (2) as follows:

$$\text{sumLB}'(u) = \sum_{j=1}^i \left| d(q, p_j) - d_{G'}(u, p_j) \cdot \alpha_t \right|. \quad (7)$$

Our search algorithm is as follows. We start with a set of candidate nodes  $\mathcal{C}$ , which is initially  $\mathbb{U}$ , and set  $\text{SumLB}(u) = 0$  for all  $u \in \mathbb{U}$ . Then, we choose a node  $p \in \mathcal{C}$  minimizing  $\text{SumLB}'$  (Eq. (7)) and remove it from  $\mathcal{C}$ . Note that

---

*t*-AESA (Query  $q$ , Radius  $r$ ,  $t$ -Spanner  $G'$ )

1.  $\mathcal{C} \leftarrow \mathbb{U}$ ,  $\alpha_t \leftarrow \frac{2/t+1}{3}$
  2. **For each**  $p \in \mathcal{C}$  **Do**  $SumLB'(p) \leftarrow 0$
  3. **While**  $\mathcal{C} \neq \emptyset$  **Do**
  4.      $p \leftarrow \operatorname{argmin}_{c \in \mathcal{C}} SumLB'(c)$ ,  $\mathcal{C} \leftarrow \mathcal{C} - \{p\}$
  5.      $d_{qp} \leftarrow d(q, p)$ , **If**  $d_{qp} \leq r$  **Then** Report  $p$
  6.      $d_{G'} \leftarrow \mathbf{Dijkstra}(G', p, t(d_{qp} + r))$
  7.     **For each**  $u \in \mathcal{C}$  **Do**
  8.         **If**  $d_{G'}(u, p) \notin [d_{qp} - r, t(d_{qp} + r)]$  **Then**  $\mathcal{C} \leftarrow \mathcal{C} - \{u\}$
  9.         **Else**  $SumLB'(u) \leftarrow SumLB'(u) + |d_{qp} - d_{G'}(u, p)| \cdot \alpha_t$
- 

Fig. 3. Our search algorithm (*t*-AESA).  $\mathbf{Dijkstra}(G', p, x)$  computes distances over the  $t$ -spanner  $G'$  from  $p$  to all nodes up to distance  $x$ , and marks the remaining ones as “farther away”.

the first object  $p = p_1$  is chosen at random. We measure  $d_{qp} \leftarrow d(q, p)$  and report  $p$  if  $d_{qp} \leq r$ . Now, we run Dijkstra’s shortest path algorithm on the  $t$ -spanner starting at  $p$ , until retrieving the first node  $v$  whose shortest-path distance to  $p$  satisfies  $d_{G'}(v, p) > t(d_{qp} + r)$ . Since Dijkstra’s algorithm gives the distances to  $p$  in increasing order, we know that all the remaining nodes will be farther away. By the  $t$ -discarding theorem, we remove from  $\mathcal{C}$  nodes  $u$  which satisfy either Eq. (5) or (6). For the non-discarded nodes we update  $sumLB'$  according to Eq (7). We repeat these steps until  $\mathcal{C} = \emptyset$ . Fig. 3 depicts the algorithm.

The number of distance evaluations performed by AESA is in practice close to  $O(1)$ , and the extra CPU time is close to  $O(n)$  [40] (the constants hide the dimensionality of the space). In our case, however, we have the additional cost of computing shortest paths. Albeit we are interested only in the nodes belonging to  $\mathcal{C}$ , we need to compute distances to many others in order to obtain those we need. We remark that the shortest-path algorithm works only up to the point where the next closest element found is far enough. In the worst case, if Dijkstra’s algorithm uses a heap data structure to find the next closest node, the total extra CPU time is  $O(n_p m \log n)$ , where  $n_p$  is the amount of nodes used as pivots, and  $m = |E| = n^{1+O(\frac{1}{t-1})}$ . Hence, the CPU time complexity of a query is at most AESA CPU time cost multiplied by  $O(n^{O(\frac{1}{t-1})} \log n)$ . As we show later, this is in practice around  $O(n^{0.13} \log n)$ , which is rather moderate.

Recall that we focus on applications where the cost to compute  $d$  dominates even heavy extra CPU costs. There are many metric spaces where computing a distance is highly expensive. For instance, in the metric space of documents under the cosine distance [5], computing a distance requires numerous disk accesses (in order to load the vector representation of the document) and hundreds of thousands of basic arithmetic operations (in order to compute the angle between the vectors representing the documents). In these cases

the distance evaluation takes several milliseconds, which is rather expensive, even compared to Dijkstra’s algorithm computations we introduced by using a  $t$ -spanner as the metric database representation.

We finish with a note on the  $t$  values of interest. In most metric spaces of practical applicability the distance histogram is concentrated, so most elements of  $\mathbb{U}$  are inside the gray ring of Fig. 2(a). By using values  $t > 1.0$ , that ring is widened, as shown in Fig. 2(b). Thus, for searching purposes, we wish to use values of  $t$  close to 1.0. In fact, if the histogram of distances is too concentrated, any value of  $t > 1.0$  could fail to discard any object. On the other hand, values of  $t$  close to 1.0 could force us to retain in the  $t$ -spanner almost every edge from the full graph. Since we want to store few edges, a trade-off between space and search time arises. This trade-off is controlled by the parameter  $t$ . In the experimental study we achieve good results with values  $t \in (1.0, 2.0]$ . However, in some metric spaces it is possible to obtain reasonable results even using values of  $t$  beyond 2.0.

#### 4 Practical $t$ -Spanner Construction Algorithms

In Section 4.1 we present a basic  $t$ -spanner construction technique. This algorithm has important deficiencies: excessive edge insertion cost and too high memory requirements. We seek practical algorithms that allow building appropriate  $t$ -spanners for metric spaces, that is, with  $t \leq 2.0$ , for complete graphs, and taking advantage of the triangle inequality. For this sake, we propose four  $t$ -spanner construction algorithms, with the goals of decreasing CPU and memory cost, and producing  $t$ -spanners of good quality (that is, with few edges). Our four algorithms are:

- (1) An optimized basic algorithm, where we limit the propagation of computations when a new edge is inserted.
- (2) A massive edge-insertion algorithm, where we amortize the cost of re-computing distances over many edge insertions.
- (3) An incremental algorithm, where nodes are added one by one to a well-formed growing  $t$ -spanner.
- (4) A recursive algorithm, combining a divide and conquer technique with a variant of the incremental algorithm.

Table 1 shows the worst case complexities obtained. Empirically, the time costs are around of the form  $C_c \cdot n^{2.24}$ , and the number of edges are around of the form  $m = C_e \cdot n^{1.13}$ , for constants  $C_c$  and  $C_e$ . This shows that good-quality  $t$ -spanners can be built in reasonable time: note that just scanning all the edges of the complete graph needs  $O(n^2)$  time.

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

Table 1

$t$ -Spanner construction worst-case complexities. The value  $k \leq n$  refers to the number of nodes that have to be checked when updating distances due to a new inserted edge, and  $m$  is the number of edges in the resulting  $t$ -spanner.

We remark that, although the  $m$  values in each row of Table 1 are different (because they are the number of edges obtained with different algorithms), our experimental results show that they are very similar in all cases.

We take no particular advantage of the metric properties of the edge weights, so our algorithms can be used on general graphs too. The only extra work needed is to precompute the shortest path among every pair of nodes, which is free when the triangle inequality holds.

From now on, we use the following terms to simplify the exposition:

- Given a pair  $u, v \in \mathbb{U}$ , the  **$t$ -condition** is  $d_{G'}(u, v) \leq t \cdot d(u, v)$ , where  $d_{G'}(u, v)$  is the distance between  $u$  and  $v$  estimated over the graph, and  $d(u, v)$  is the distance in the metric space.
- We say that a distance  $d(u, v)$  is  **$t$ -estimated**, if it fulfills the  $t$ -condition on the graph under consideration. Otherwise, it is **not  $t$ -estimated**.

#### 4.1 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 zero, except for the distances between each node and itself ( $d(u, u) = 0$ ). Edges are inserted successively until all the distance estimations satisfy 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.

The algorithm uses two symmetric matrices. The first, *real*, contains the true

---

**$t$ -Spanner0** (Stretch  $t$ , Vertices  $\mathbb{U}$ )

1. **For each**  $u, v \in \mathbb{U}$  **Do**
2.      $real(u, v) \leftarrow d(u, v)$
3.     **If**  $u = v$  **Then**  $estim(u, v) \leftarrow 0$  **Else**  $estim(u, v) \leftarrow \infty$
4.  $t$ -Spanner  $\leftarrow \emptyset$  //  $t$ -spanner edge structure
5. **For each**  $e = (e_u, e_v) \in real$  **Do** // chosen in increasing cost order
6.     **If**  $estim(e) > t \cdot real(e)$  **Then** //  $e$  is not  $t$ -estimated
7.      $t$ -Spanner  $\leftarrow t$ -Spanner  $\cup \{e\}$
8.     **For each**  $v_i, v_j \in \mathbb{U}$  **Do**
9.          $d_1 \leftarrow estim(v_i, e_u) + estim(e_v, v_j)$
10.         $d_2 \leftarrow estim(v_j, e_u) + estim(e_v, v_i)$
11.         $estim(v_i, v_j) \leftarrow \min(estim(v_i, v_j), \min(d_1, d_2) + real(e))$

---

Fig. 4. Basic  $t$ -spanner construction algorithm ( $t$ -Spanner 0).

distances between all the objects, and the second,  $estim$ , contains the distance estimations obtained with the  $t$ -spanner under construction. The  $t$ -spanner is stored as 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 updating mechanism is similar to the distance calculation mechanism of Floyd’s algorithm [22], except that edges, not nodes, are inserted into the set. Upon insertion of an edge  $e = (e_u, e_v)$ , every path from  $v_i$  to  $v_j$  in  $\mathbb{U}$  can now go through edge  $e$ . There are two choices for this:  $v_i \leftrightarrow e_u \leftrightarrow e_v \leftrightarrow v_j$  and  $v_i \leftrightarrow e_v \leftrightarrow e_u \leftrightarrow v_j$ . Fig. 4 depicts the basic  $t$ -spanner construction algorithm.

**Analysis.** The basic  $t$ -spanner construction algorithm takes  $\Theta(n^2)$  distance evaluations, just as AESA index construction,  $\Theta(mn^2)$  CPU time (recall that  $n = |V|$  and  $m = |E|$ ), and  $\Theta(n^2 + m) = \Theta(n^2)$  memory. Its main deficiencies are excessive edge insertion cost and too high memory requirements.

#### 4.2 Optimized Basic Algorithm

Like the basic algorithm (Section 4.1), this algorithm makes use of  $real$  and  $estim$  matrices, choosing the edges in increasing weight order. The optimization focuses on the mechanism to update distance estimations.

The main idea is to control the propagation of the computation, updating only the distance estimations that can be affected by the insertion of the new edge, and disregarding those that can be proved not to be affected by the new

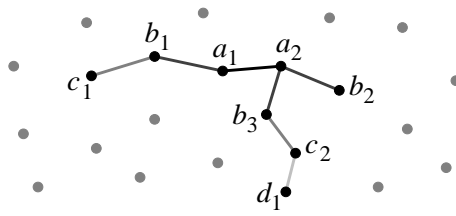


Fig. 5. Propagation of distance estimations.

inserted edge.

Fig. 5 illustrates the insertion of a new edge  $(a_1, a_2)$ . Note that it is necessary to propagate the computation only to the nodes that improve their distance estimation to  $a_1$  or  $a_2$ . In the first step we update only the cost of the edge that was inserted. The computation then propagates to the neighbors of the  $a_i$  nodes, namely nodes  $\{b_1, b_2, b_3\}$ , then to nodes  $\{c_1, c_2\}$ , and finally  $\{d_1\}$ . The propagation stops when a node does not improve its current distance estimations or when it does not have further neighbors.

The complete algorithm reviews all the edges of the graph. For each edge, it iterates until no further propagation is necessary. In order to control the propagation, the algorithm uses two sets, *ok* and *check*.

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

Fig. 6 depicts the optimized basic algorithm.

**Analysis.** The optimized basic algorithm takes  $\Theta(n^2)$  distance evaluations and  $\Theta(mk^2 + n^2)$  CPU time, where  $k$  is the number of neighbors to check when inserting an edge (that is, final size of set *ok*). In the worst case this becomes  $\Theta(mn^2)$ , just as the basic algorithm, but on the average it is much less. The memory usage is  $\Theta(n^2 + m) = \Theta(n^2)$ . Although this algorithm reduces the CPU time, this time is still high, and the memory requirements are also too high.

A good feature of the algorithm is that, just like the basic algorithm, it produces good-quality  $t$ -spanners (few edges). We have used its results to predict the expected number of edges per node in a  $t$ -spanner, so as to fine-tune other algorithms that rely on massive insertion of edges. We call  $|E_{t\text{-Spanner } 1}(n, dim, t)|$  the expected number of edges in a metric space of  $n$  objects, intrinsic dimension  $dim$ , and stretch  $t$ . In order to determine the intrinsic dimension of a given metric space we develop an empirical method, which will be described

---

```

t-Spanner1 (Stretch  $t$ , Vertices  $\mathbb{U}$ )
1.  For each  $u, v \in \mathbb{U}$  Do
2.       $real(u, v) \leftarrow d(u, v)$ 
3.      If  $u = v$  Then  $estim(u, v) \leftarrow 0$  Else  $estim(u, v) \leftarrow \infty$ 
4.   $t$ -Spanner  $\leftarrow \emptyset$  //  $t$ -spanner edge structure
5.  For each  $e = (e_u, e_v) \in real$  Do // chosen in increasing cost order
6.      If  $estim(e) > t \cdot real(e)$  Then //  $e$  is not  $t$ -estimated
7.           $t$ -Spanner  $\leftarrow t$ -Spanner  $\cup \{e\}$ ,  $estim(e) \leftarrow real(e)$ 
8.           $ok \leftarrow \{e_u, e_v\}$ ,  $check \leftarrow \mathbf{adjacency}(ok) - ok$ 
9.          For each  $c \in check$  Do
10.             If  $estim(c, e_v) + real(e) < estim(c, e_u)$  or
11.              $estim(c, e_u) + real(e) < estim(c, e_v)$  Then
12.                 For each  $o \in ok$  Do
13.                      $d_1 \leftarrow estim(c, e_u) + estim(e_v, o)$ 
14.                      $d_2 \leftarrow estim(c, e_v) + estim(e_u, o)$ 
15.                      $estim(c, o) \leftarrow \min(estim(c, o), \min(d_1, d_2) + real(e))$ 
16.                      $check \leftarrow check \cup (\mathbf{adjacency}(c) - ok)$ 
17.                      $ok \leftarrow ok \cup \{c\}$ ,  $check \leftarrow check - \{c\}$ 

```

---

Fig. 6. Optimized basic algorithm ( $t$ -Spanner 1).

in Section 5.1.1. In Section 5 (Tables 2 and 3) we show some estimations obtained.

### 4.3 Massive Edge-Insertion Algorithm

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

Since we insert edges less carefully than before, the resulting  $t$ -spanner could be of lower quality. Our effort aims at minimizing this effect.

The algorithm has three stages. In the first stage, it builds the  $t$ -spanner backbone by inserting whole minimum spanning trees (MSTs), and determines the global list of not  $t$ -estimated edges (*pending*). In the second stage, it refines the  $t$ -spanner by adding more edges to improve the not  $t$ -estimated edges. In the third stage, it inserts all the remaining “hard” edges.

The algorithm uses two heuristic values:

$H_1$  predicts the expected number of edges per node, and it is obtained from the optimized basic algorithm edge model:  $H_1 = |E_{t\text{-Spanner } 1}(n, dim, t)|/n$ . With  $H_1$  we will define thresholds to determine whether or not to insert the remaining edges (those still not  $t$ -estimated) of the current node. Note that  $|E_{t\text{-Spanner } 1}(n, dim, t)|$  is an optimistic predictor of the resulting  $t$ -spanner size using the massive edge-insertion algorithm, so we can use  $H_1$  as an expected lower bound of the number of edges per node. As it is shown by experimental results in Section 5, in most cases the sizes of resulting  $t$ -spanners produced by any of our algorithms are rather similar, which implies that  $n \cdot H_1$  is quite a good predictor for the expected size of the  $t$ -spanner.

$H_2$  is used to control the growth of the *pending* list size and will give a criterion to decide when to insert an additional MST into the  $t$ -spanner under construction. The maximum *pending* list size is  $H_2 \cdot |t\text{-Spanner}|$ , where  $t\text{-Spanner}$  is  $t$ -spanner under construction. After preliminary experiments we have fixed  $H_2 = 1.2$ . With values lower than 1.2 the algorithm takes more processing time without improving the number of edges, and with higher values the algorithm takes less processing time, but it inserts more edges than necessary and needs more memory to build the  $t$ -spanner.

We describe now the stages of the algorithm.

**Stage 1.** We insert successive MSTs to the  $t$ -spanner. The first MST follows the basic Prim's algorithm [35], but next MSTs are built choosing only edges that have not yet been inserted.

We make a single pass over the nodes, adding to the *pending* list the not  $t$ -estimated edges that are incident upon the current node under revision. Every time the size of list *pending* exceeds  $H_2$ , we insert a new MST (recall that  $H_2$  depends on the current  $t$ -spanner size,  $|t\text{-Spanner}|$ ). Notice that, the more MSTs are inserted, the tighter the estimations of the distance (by using the shortest path between two objects). Thus, after an MST is inserted, we remove the edges from *pending* list that become  $t$ -estimated. Additionally, if the current node has no more than  $H_1/2$  pending edges, we just insert them, since we only need a small set of edges in order to fix all the distance estimations for this node.

This stage continues until we pass through all the nodes. The output is the  $t$ -spanner backbone and the global list of pending edges (*pending*).

**Stage 2.** In the second stage we reduce the *pending* list. For this sake, we traverse the list of nodes with pending edges (*pendingNodes*), from those with most to those with least pending edges. For each such node, we check which edges have to improve their  $t$ -estimation and which do not. Note that edges originally in the pending list may have become well  $t$ -estimated along the process due to new added edges. We need to insert more edges in order to improve distance estimations (of edges that are incident to the current node that remain in *pending*), so from those still not  $t$ -estimated edges, we insert  $H_1/4$  smallest-cost edges, and proceed to the next node.

Note that the current node may have more than  $H_1/4$  not  $t$ -estimated edges. However, this technique allows us to review in the first place nodes that require more attention, without concentrating all the efforts in the same node. With values lower than  $H_1/4$  the algorithm takes more processing time without improving the number of edges considerably, and with higher values the algorithm inserts more edges than necessary.

The process considers two special cases. The first case occurs when we have inserted more than  $n$  edges overall, in which case we regenerate and re-sort list *pendingNodes* and restart the process (so that we resume the work on nodes with many pending edges). The second special case occurs when the pending list of the current node is so small (less than  $H_1/4$  edges) that we simply insert all its elements.

The condition to finish the second stage is that the *pending* list size is smaller than  $n/2$ , since these hard-to-estimate edges are so few that it is not worth the effort of reducing the pending list once again. We made preliminary experiments in order to fix this value, and obtained the best trade-off between CPU time and  $t$ -spanner size with  $n/2$ .

**Stage 3.** The hard-to-estimate edges remain in the *pending* list, so we just insert the *pending* list into the  $t$ -Spanner to complete the  $t$ -spanner construction.

Fig. 7 depicts the massive edge-insertion algorithm.

**Analysis.** The massive edge-insertion algorithm takes  $\Theta(nm)$  distance evaluations,  $\Theta(nm \log n)$  CPU time, and  $\Theta(n + m) = \Theta(m)$  memory. It is easy to see that the space requirement is  $\Theta(m)$ : the *pending* list is never larger than  $\Theta(m)$  because at each iteration of stage 1 it grows at most by  $n - 1$ , and as soon as it becomes larger than  $H_2 \cdot |t\text{-Spanner}| \leq H_2 \cdot m$  we add a new MST into  $t\text{-Spanner}$  (so  $m$  grows by  $n - 1$  as well). The distance evaluations come from running a  $\Theta(n^2)$ -cost Prim's algorithm at most  $m/n$

times at stage 1 (since each run adds  $n - 1$  edges). The CPU time comes from running a  $\Theta(m \log n)$ -time Dijkstra’s algorithm once per node, and a  $\Theta(n^2)$ -time Prim’s algorithm at most  $m/n$  times at stage 1. This adds up  $\Theta(nm \log n) + \Theta(mn) = \Theta(nm \log n)$  CPU time. At stage 2 we insert edges in groups of  $\Theta(m/n)$ <sup>1</sup>, running Dijkstra’s algorithm after each insertion, until we have inserted  $|pending| - n/2 = \Theta(m)$  edges overall. This accounts for other  $n$  times we run the  $\Theta(m \log n)$  Dijkstra’s algorithm, adding another  $\Theta(nm \log n)$  term.

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

#### 4.4 Incremental Node Insertion Algorithm

This algorithm reduces the amount of distance evaluations to just  $n(n - 1)/2$ , while preserving the amortized update cost idea. We insert nodes one by one, not edges. The invariant is that, for nodes  $1 \dots i - 1$ , we have a well-formed  $t$ -spanner, and we want to insert the  $i$ -th node to the growing  $t$ -spanner.

This algorithm, unlike the previous ones, makes a local analysis of nodes and edges, that is, it takes decisions before having knowledge of the whole edge set. This can affect the quality of the  $t$ -spanner.

For each new node  $i$ , the algorithm carries out two operations: the first is to connect the node  $i$  to the growing  $t$ -spanner using the cheapest edge towards the previous nodes; the second is to verify that its distance estimations satisfy the  $t$ -condition, adding some edges to node  $i$  until the invariant is restored. We repeat this process until the whole node set is inserted.

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 insert  $\delta = H_1(i, dim, t)/5 = \frac{|E_{t\text{-Spanner } 1}(i, dim, t)|}{5i}$  edges at a time, in order to reduce the processing time. The factor 5 in the denominator is tuned so that inserting more edges at a time obtains lower processing time but the size of the  $t$ -spanner is increased; whereas inserting less edges at a time increases the processing time without significantly reducing the  $t$ -spanner size.

For the verification of distances to the new node we use an incremental Dijkstra’s algorithm with limited propagation, that is, instead of assuming that distances have an initial value  $\infty$ , the algorithm receives the current computed distances in an array. This is because, in Dijkstra’s algorithm, edges incident

<sup>1</sup> This assumes that  $|E_{t\text{-Spanner } 1}| \approx |E_{t\text{-Spanner } 2}|$ , which is experimentally verified in Section 5.

---

**$t$ -Spanner2** (Stretch  $t$ , Vertices  $\mathbb{U}$ )

1.  $t\text{-Spanner} \leftarrow \text{MST}$  //  $t$ -spanner edge structure, initially has the first MST
2.  $\text{pending} \leftarrow \emptyset$  // global pending edge list
3.  $H_1 \leftarrow |E_{t\text{-Spanner1}}(n, \text{dim}, t)|/n$

// Stage 1: Generating the  $t$ -Spanner backbone and pending list

4. **For each**  $u \in \mathbb{U}$  **Do**
5.     **If**  $|\text{pending}| > H_2 \cdot |t\text{-Spanner}|$  **Then**
6.          $t\text{-Spanner} \leftarrow t\text{-Spanner} \cup \text{MST}$  // built over the non-inserted edges
7.      $d_{G'} \leftarrow \text{Dijkstra}(t\text{-Spanner}, u)$  //  $d_{G'}(u, v) = d_{t\text{-Spanner}}(u, v)$
8.     **For each**  $v \in \mathbb{U}$  **Do**
9.         **If**  $d_{G'}(u, v) \leq t \cdot d(u, v)$  **Then**
10.             **If**  $(u, v) \in \text{pending}$  **Then**  $\text{pending} \leftarrow \text{pending} - \{(u, v)\}$
11.             **Else**  $\text{pending} \leftarrow \text{pending} \cup \{(u, v)\}$
12.     **If**  $|\text{pending}(u)| \leq H_1/2$  **Then**
13.         // Let  $\text{pending}(u) = \{e \in \text{pending}, \exists v, e = (u, v)\}$
14.          $t\text{-Spanner} \leftarrow t\text{-Spanner} \cup \text{pending}(u)$
15.          $\text{pending} \leftarrow \text{pending} - \text{pending}(u)$

// Stage 2: Reducing pending

16. **While**  $|\text{pending}| > n/2$  **Do**
17.      $\text{pendingNodes} \leftarrow$  nodes sorted by decreasing  $|\text{pending}(u)|$
18.     **For each**  $u \in \text{pendingNodes}$  **Do**
19.         **If** more than  $n$  edges have been inserted **Then** // special case 1
20.             break
21.         **If**  $|\text{pending}(u)| < H_1/4$  **Then** // special case 2
22.              $t\text{-Spanner} \leftarrow t\text{-Spanner} \cup \text{pending}(u)$
23.              $\text{pending} \leftarrow \text{pending} - \text{pending}(u)$
24.         **Else**
25.              $d_{G'} \leftarrow \text{Dijkstra}(t\text{-Spanner}, u)$
26.             **For each**  $v \in \text{pending}(u)$  **Do** **If**  $d_{G'}(u, v) \leq t \cdot d(u, v)$  **Then**
27.                  $\text{pending} \leftarrow \text{pending} - \{(u, v)\}$
28.              $\text{smallest} \leftarrow H_1/4$  smallest edges in  $\text{pending}(u)$
29.              $t\text{-Spanner} \leftarrow t\text{-Spanner} \cup \text{smallest}$
30.              $\text{pending} \leftarrow \text{pending} - \text{smallest}$

// Stage 3: Inserting the hard edges

31.  $t\text{-Spanner} \leftarrow t\text{-Spanner} \cup \text{pending}$

---

Fig. 7. Massive edge-insertion algorithm ( $t$ -Spanner 2).

upon a processed node are considered only if the node has improved its current distance estimation. Furthermore, note that when the distance from node  $i$  to node  $j$  is not  $t$ -estimated, we do not really need to know how poorly estimated it is. Then, in the first iteration we initialize the distance estimation values to be just slightly not  $t$ -estimated:  $t \cdot d(u_i, u_j) + \varepsilon$  for  $j \in [1 \dots i - 1]$ , where  $\varepsilon$  is a small positive constant. For the next iterations, Dijkstra's algorithm

---

**$t$ -Spanner3** (Stretch  $t$ , Vertices  $\mathbb{U}$ )

1.  $t$ -Spanner  $\leftarrow \emptyset$  //  $t$ -spanner edge structure
  2. **For**  $i \in [1, n]$  **Do**
  3.      $\delta \leftarrow \frac{|E_{t\text{-Spanner1}}(i, \dim, t)|}{5^i}$  // incremental  $H_1$
  4.      $u \leftarrow \text{node}_i, k \leftarrow \text{argmin}_{j \in [1, i-1]} \{d(u, \text{node}_j)\}$
  5.      $t$ -Spanner  $\leftarrow t$ -Spanner  $\cup \{(u, \text{node}_k)\}$  // inserting the cheapest edge
  6.      $d_{G'} \leftarrow \{( \text{node}_j, t \cdot d(u, \text{node}_j) + \varepsilon), j \in [1 \dots i - 1]\}$  // propagation limit
  7.     **Do**
  8.          $d_{G'} \leftarrow \text{Dijkstra}(t\text{-Spanner}, u, d_{G'})$  // incremental Dijkstra
  9.          $\text{pending} \leftarrow \{(u, \text{node}_j), d_{G'}(u, \text{node}_j) > t \cdot d(u, \text{node}_j), j < i\}$
  10.          $\text{smallest} \leftarrow \delta$  cheapest edges in  $\text{pending}$
  11.          $t$ -Spanner  $\leftarrow t$ -Spanner  $\cup \text{smallest}$
  12.     **While**  $\text{pending} \neq \emptyset$
- 

Fig. 8. Incremental node insertion algorithm ( $t$ -Spanner 3).

reuses the array previously computed, because there is no need to propagate distances from nodes whose estimations have not improved. Note that this allows the shortest path propagation to stop as soon as possible, both in the first and in the following iterations.

Fig. 8 depicts the incremental node insertion algorithm.

**Analysis.** The incremental node insertion algorithm takes  $\Theta(n^2)$  distance evaluations,  $\Theta(nm \log n)$  CPU time, and  $\Theta(n + m) = \Theta(m)$  memory. The CPU time comes from the fact that every node runs Dijkstra’s algorithm  $\frac{m}{n}/\delta = \Theta(1)$  times (see Footnote 1).

#### 4.5 Recursive Algorithm

The incremental algorithm is an efficient approach to construct metric  $t$ -spanners, but it does not perform a global edge analysis. A way to solve this is to try that the set in which the  $t$ -spanner is incrementally built is made up of objects close to each other. Following this principle, we present a solution that divides the object set into two compact subsets, recursively builds sub- $t$ -spanners for the subsets, and then merges them.

For the initial set division we take two objects  $p_1$  and  $p_2$  far away from each other, which we call *representatives*, and divide the set among objects nearer to  $p_1$  and nearer to  $p_2$ . Fig. 9(a) illustrates. For the recursive divisions we reuse the corresponding representative, and the element farthest to it becomes the other. The recursion finishes when we have less than 3 objects.

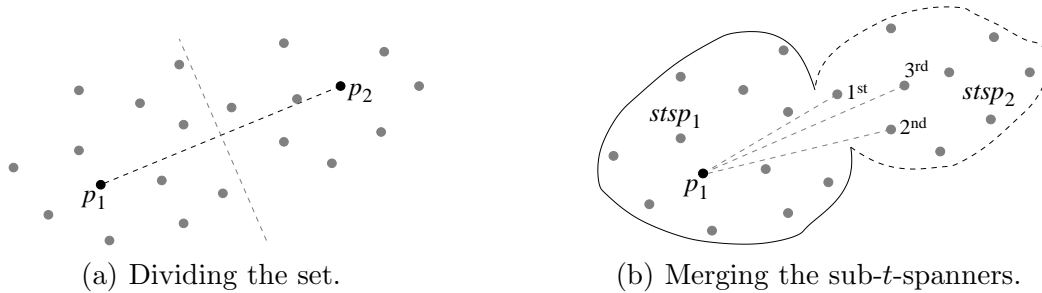


Fig. 9. In (a), we select  $p_1$  and  $p_2$ , and then divide the set. In (b), the merging step chooses objects according to their distances towards  $p_1$ .

The merging 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  $u$  closest to  $p_1$ , and insert it into the sub- $t$ -spanner represented by  $p_1$  ( $stsp_1$ ) verifying that all the distances towards  $V_1$  are  $t$ -estimated. Note that this is equivalent to using 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 nodes in  $stsp_2$  are inserted into  $stsp_1$ . Fig. 9(b) illustrates. Note that edges already present in  $stsp_2$  are conserved. We also use the  $H_1$  heuristic, but this time we recompute  $H_1$  at the beginning of the merging step (not upon inserting every node). We insert  $\delta = H_1(|nodes|, dim, t)/5 = \frac{|E_{t\text{-Spanner}}(|nodes|, dim, t)|}{5|nodes|}$  edges at a time, where  $nodes = V_1 \cup V_2$ , in order to reduce the processing time.

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). Albeit we are interested only in the nodes belonging to  $stsp_1$ , we need also to compute distances to  $stsp_2$  to obtain those we need. 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  $\infty$  for  $(u_i, u_j) \in V_2 \times V_2$ , where  $\varepsilon$  is a small positive constant. For the next iterations, Dijkstra's algorithm reuses the previously computed array.

Fig. 10 depicts the recursive algorithm and the auxiliary functions used to build and merge sub- $t$ -spanners.

**Analysis.** The recursive algorithm requires  $\Theta(n^2)$  distance evaluations,  $\Theta(nm \log n)$  CPU time, and  $\Theta(n + m) = \Theta(m)$  memory. The cost of dividing sets does not affect that of the underlying incremental construction.

---

***t*-Spanner4** (Stretch  $t$ , Vertices  $\mathbb{U}$ )

1.  $t$ -Spanner  $\leftarrow \emptyset$  //  $t$ -spanner edge structure
2.  $(p_1, p_2) \leftarrow$  two distant objects
3.  $(V_1, V_2) \leftarrow \mathbb{U}$  divided according to distances towards  $(p_1, p_2)$
4.  $stsp_1 \leftarrow \mathbf{makeSubtSpanner}(p_1, V_1)$ ,  $stsp_2 \leftarrow \mathbf{makeSubtSpanner}(p_2, V_2)$
5.  $t$ -Spanner  $\leftarrow \mathbf{mergeSubtSpanner}(stsp_1, stsp_2)$

**makeSubtSpanner**(Representative  $p$ , Vertices  $V$ )

6. **If**  $|V| = 1$  **Then Return**  $t$ -Spanner( $V = \{p\}, \emptyset$ )
7. **Else If**  $|V| = 2$  **Then Return**  $t$ -Spanner( $V = \{p, v\}, \{(p, v)\}$ )
8. **Else**
9.      $p_{remote} \leftarrow \operatorname{argmax}_{v \in V} \{d(p, v)\}$
10.     $(V, V_{remote}) \leftarrow V$  divided according to distances towards  $(p, p_{remote})$
11.     $stsp_p \leftarrow \mathbf{makeSubtSpanner}(p, V)$
12.     $stsp_{remote} \leftarrow \mathbf{makeSubtSpanner}(p_{remote}, V_{remote})$
13.    **Return**  $\mathbf{mergeSubtSpanner}(stsp_p, stsp_{remote})$

**mergeSubtSpanner** ( $t$ -Spanner  $stsp_1$ ,  $t$ -Spanner  $stsp_2$ )

14. **If**  $|\mathbf{nodes}(stsp_1)| \leq |\mathbf{nodes}(stsp_2)|$  **Then**  $stsp_1 \leftrightarrow stsp_2$
15.  $nodes \leftarrow \mathbf{nodes}(stsp_1) \cup \mathbf{nodes}(stsp_2)$
16.  $edges \leftarrow \mathbf{edges}(stsp_1) \cup \mathbf{edges}(stsp_2)$
17.  $\delta \leftarrow \frac{|E_{t\text{-Spanner1}}(|nodes|, dim, t)|}{5|nodes|}$  // incremental  $H_1$
18.  $p_1 \leftarrow \mathbf{Representative}(stsp_1)$
19. **For each**  $u \in \mathbf{nodes}(stsp_2)$  in increasing order of  $d(u, p_1)$  **Do**
20.     // defining the propagation limit towards  $stsp_1$
21.     **For each**  $v \in \mathbf{nodes}(stsp_1)$  **Do**  $d_{G'}(u, v) \leftarrow t \cdot d(u, v) + \varepsilon$
22.     **For each**  $v \in \mathbf{nodes}(stsp_2)$  **Do**  $d_{G'}(u, v) \leftarrow \infty$
23.     **Do**
24.          $d_{G'} \leftarrow \mathbf{Dijkstra}(edges, u, d_{G'})$  // incremental Dijkstra
25.          $pending \leftarrow \{(u, v), d_{G'}(u, v) > t \cdot d(u, v), v \in stsp_1\}$
26.          $smallest \leftarrow \delta$  cheapest edges in  $pending$
27.          $edges \leftarrow edges \cup smallest$
28.     **While**  $pending \neq \emptyset$
29. **Return**  $t$ -Spanner( $nodes, edges$ )

---

Fig. 10. Recursive algorithm ( $t$ -Spanner 4).

## 5 Experimental Results

We have tested our construction and search algorithms on spaces of vectors, strings and documents (these last two are of interest to Information Retrieval applications [5]). The experiments were run on an Intel Pentium IV of 2 GHz, with 2.0 GB of RAM, with local disk, under SuSE Linux 7.3 operating system,

with kernel 2.4.10-4GB i686, using g++ compiler version 2.95.3 with optimization option `-O9`, and the processing time measured user time. For construction algorithms, we are interested in measuring the CPU time needed and the amount of edges generated by each algorithm (the number of distance evaluations is always  $n(n - 1)/2$  in the competitive alternatives). For the search algorithm, we are interested in measuring the number of distance evaluations performed in the retrieval operation. For shortness we have called the optimized basic algorithm *t*-Spanner 1, the massive edge-insertion algorithm *t*-Spanner 2, the incremental algorithm *t*-Spanner 3, the recursive algorithm *t*-Spanner 4, and the simulated AESA over the *t*-spanner *t*-AESA.

The construction experiments compare *t*-Spanner 1, 2, 3 and 4, in order to determine which is the most appropriate to metric spaces, with  $t \in [1.4, 2.0]$ . The search experiments use *t*-spanners with stretch factors  $t \in [1.4, 2.0]$ , and compare them against AESA. Since *t*-spanners offer a time-space trade-off and AESA does not, we also consider pivot-based indexes with varying number of pivots. For every *t* value, we measure the size of the resulting *t*-spanner and build a pivot-based index using the same amount of memory (pivots are chosen at random). This way we compare *t*-spanners against the classical space-time trade-off for AESA. Note that with values of  $t < 1.4$  the performance of our construction algorithms noticeably degrades, whereas the search algorithm does not improve considerably.

Since in some cases the pivots were too many compared to the average number of candidates to eliminate, we decided to stop using further pivots when the remaining candidates were fewer than the remaining pivots. This way we try not to pay more for having more available pivots than necessary. Also, it turns out that, sometimes, even the smallest number of pivots shown is beyond the optimal. In these cases we also show results with fewer pivots, until we reach the optimum.

### 5.1 Uniformly distributed Vectors under Euclidean Distance

We start our experimental study with the space of vectors uniformly distributed in a real  $D$ -dimensional cube under the Euclidean distance, that is,  $([-1, 1]^D, L_2)$ , for  $D \in [4, 24]$ . This metric space allows us to measure the effect of the space dimension  $D$  on our algorithms. We have not explored larger  $D$  values because  $D = 28$  is already too high-dimensional: we can only build *t*-spanners in reasonable time for  $t \geq 1.8$ , which is too large for searching. We remind that all metric space search algorithms fail for these large values of  $D$ .

In particular, for *t*-Spanner 1, we can obtain an edge model to implement the  $H_1$  heuristic. This way, if we compute the intrinsic dimensionality of a given

metric space  $\mathcal{M}$ , we can apply the  $H_1$  heuristic edge model on  $\mathcal{M}$ , even if  $\mathcal{M}$  has no coordinates. For this sake, in Section 5.1.1 we show an experimental method to estimate the intrinsic dimensionality of a given metric space.

Note that we have not used the fact that the space has coordinates, but have rather treated points as abstract objects in an unknown metric space. Computing a single distance takes from 0.893 microseconds in the 4-dimensional space, to 1.479 microseconds in the 24-dimensional space.

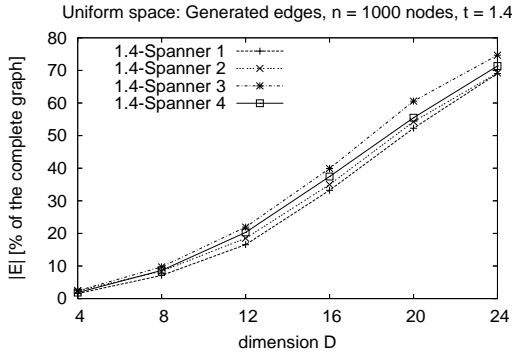
In the construction experiments, we use uniform data sets of varying size  $n \in [200 \dots 2,000]$ . We first compare the construction algorithms, in order to choose the best. Later, search experiments are carried out over the  $t$ -spanner produced by the best construction algorithm. In the search experiments, we index a uniform data set formed by 10,000 objects, and select 100 random queries not included in the index, using search radii that on average retrieve 1 object ( $r = 0.137$  for  $D = 4$ ,  $r = 0.570$  for  $D = 8$ ,  $r = 1.035$  for  $D = 12$ ,  $r = 1.433$  for  $D = 16$ ,  $r = 1.812$  for  $D = 20$  and  $r = 2.135$  for  $D = 24$ ).

### 5.1.1 Estimating the Intrinsic Dimensionality of a General Metric Space

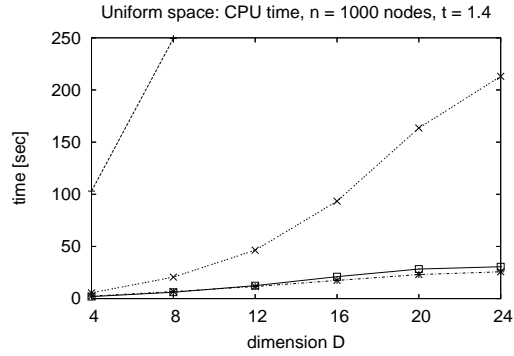
As said in the Introduction, the higher the intrinsic dimensionality  $dim$  of a given metric space  $\mathcal{M} = (\mathbb{X}, d)$ , the more difficult to solve proximity queries. Nevertheless, there is not an accepted criterion to measure  $dim$ . Yet, most authors agree in that the intrinsic dimensionality of a  $D$ -dimensional vector space with uniform distribution is simply  $D$  [43,16].

Hence, to experimentally estimate the intrinsic dimensionality of a given metric space  $\mathcal{M}$ , we approximate  $dim$  with the dimensionality  $D$  of the uniformly distributed vector space that performs similarly to  $\mathcal{M}$  for a given benchmark comparison. Later, we refine the estimation by exploring values of  $dim$  around  $D$ . Therefore, to compute  $H_1$  we test  $t$ -Spanner 1 in all of the metric spaces we use in the experiments in order to estimate the intrinsic dimensionality for each space. However, as  $t$ -Spanner 1 is, by far, the slowest constructing algorithm, we run the experiments to estimate  $dim$  over a small subset of the objects. Next, we refine the estimation of  $dim$  using some of the massive edge-insertion algorithms ( $t$ -spanner 2, 3 or 4).

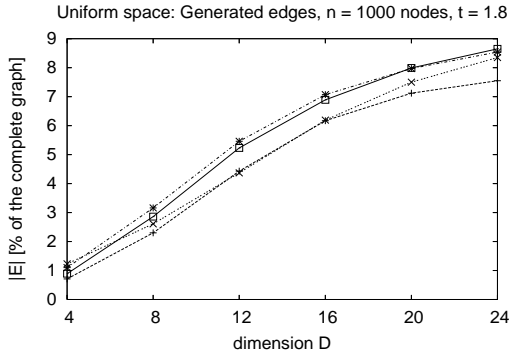
Thus, the procedure has the following stages. The first consists in computing the edge model  $|E_{t\text{-Spanner } 1}(n, D, t)|$  in  $([-1, 1]^D, L_2)$ . The second consists in testing  $t$ -Spanner 1 over a small subset of the metric space of interest  $\mathcal{M}$  and computing its edge model. Third, we find out which value  $D$  in the model for the uniform space corresponds to the performance measured in  $\mathcal{M}$ . Finally, we test some of the massive edge-insertion algorithms using  $dim$  values around  $D$  to find which  $dim$  value produces the smallest  $t$ -spanner without increasing the CPU time considerably.



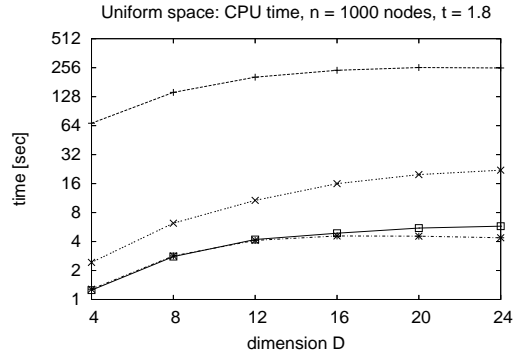
(a)  $t$ -Spanner size for  $t = 1.4$ .



(b) CPU time for construction for  $t = 1.4$ .



(c)  $t$ -Spanner size for  $t = 1.8$ .



(d) CPU time for construction for  $t = 1.8$ .

Fig. 11.  $t$ -Spanner construction in the uniform space of 1,000 nodes for  $t = 1.4$  and 1.8, as a function of  $D$ . On the left, edges generated ( $t$ -spanner quality) as a percentage of the complete graph. On the right, construction time. In (b), for  $D = 24$   $t$ -Spanner 1 reaches 1900 seconds. In (d), note the log-scale. All the plots use the legend of (a).

### 5.1.2 Construction

Fig. 11 shows a comparison among the four algorithms on the uniform data set where we vary  $D \in [4, 24]$ , for  $n = 1,000$  nodes and  $t = 1.4$  and 1.8. Figs. 11(b) and 11(d) show that  $t$ -Spanner 1 is impractically costly, but it produces the best (that is, smallest)  $t$ -spanner as shown in Figs. 11(a) and 11(c). The next slowest algorithm is  $t$ -Spanner 2, being  $t$ -Spanner 3 and 4 very similar in performance. On the other hand, the quality of the generated  $t$ -spanners is rather similar for all, being  $t$ -Spanner 3 the algorithm that produces  $t$ -spanners with most edges (arguably because of the local analysis it performs). Depending on the dimension, the next best-quality  $t$ -spanners are produced by  $t$ -Spanner 4 (low dimensions,  $D < 8$ ) or by  $t$ -Spanner 2 (medium and high dimensions,  $D \geq 8$ ). Note that the difference in  $t$ -spanner quality becomes less significant for higher dimensions. Finally, it is interesting to notice that, even in dimension  $D = 24$  and for  $t = 1.8$ , the number of edges in the resulting  $t$ -spanners is still less than 8.7% of the complete graph.

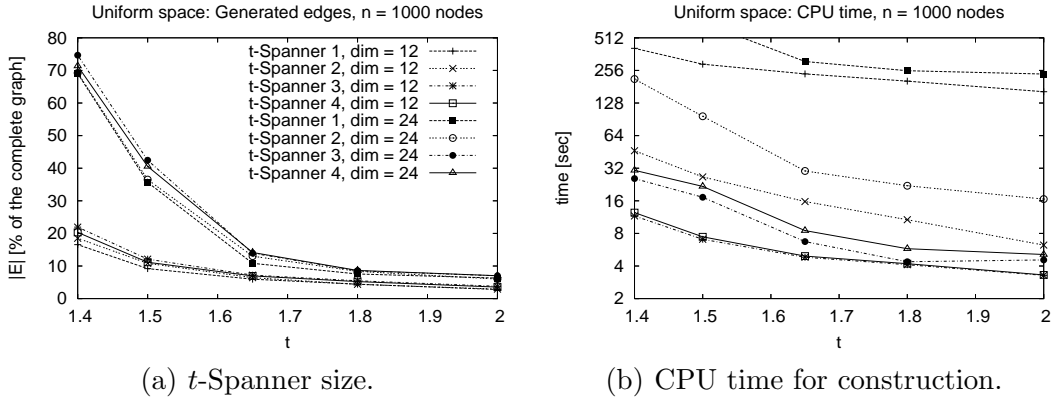


Fig. 12.  $t$ -Spanner construction in uniform spaces of 1,000 nodes for  $D = 12$  and 24, as a function of  $t$ . On the left, edges generated ( $t$ -spanner quality) as a percentage of the complete graph. On the right, construction time. In (b),  $t$ -Spanner 1 reaches 1900 seconds for  $D = 24$  and  $t = 1.4$ , also note the log-scale. (b) uses the legend of (a).

Fig. 12 compares the four algorithms where we vary  $t \in [1.4, 2.0]$ , for  $n = 1,000$  nodes and  $D = 12$  and 24. Note that, for  $t > 1.65$ , all of our algorithms produce  $t$ -spanners of good quality. In practice, to construct a 1.65-spanner we need 14% of edges of the complete graph in high dimensionality ( $D = 24$ ), 7% in medium dimensionality ( $D = 12$ ) and just 1.44% in low dimensionality spaces ( $D = 4$ ). Moreover, to construct a 2.0-spanner for the same values of  $D$ , we need just a 7%, 3.8% and 1% of the complete graph, respectively. Furthermore,  $t$ -Spanner 3 and 4 also perform well with respect to CPU time.

It is also interesting to notice that the joint effect of high dimensionality ( $D > 16$ ) and small values of  $t$  ( $t < 1.5$ ) produces a sharp increase both in the number of generated edges and the CPU time.

Fig. 13 shows the effect of the set size  $n$  in our algorithms. It can be seen that, in low dimensions ( $D = 5$ ),  $t$ -Spanner 2, 3 and 4 are slightly super-quadratic in CPU time. On the other hand, all the algorithms produce  $t$ -spanners slightly super-linear in the number of edges.

We conclude that the fastest construction algorithm for all dimensions  $D \in [4, 24]$  is  $t$ -Spanner 3, closely followed by  $t$ -Spanner 4. The other two are very far away in performance. However,  $t$ -Spanner 3 produces  $t$ -spanners with the worst quality (many edges), albeit all the qualities are indeed close. This result was expected, since its incremental methodology locally analyzes the insertion of edges. On the other hand, the recursive algorithm of  $t$ -Spanner 4 strongly improves the quality of the incremental algorithm, profiting from the global analysis of the edge set. In some cases,  $t$ -Spanner 4 is competitive even with the optimized basic algorithm in the number of generated edges, yet using 50 times less CPU time.

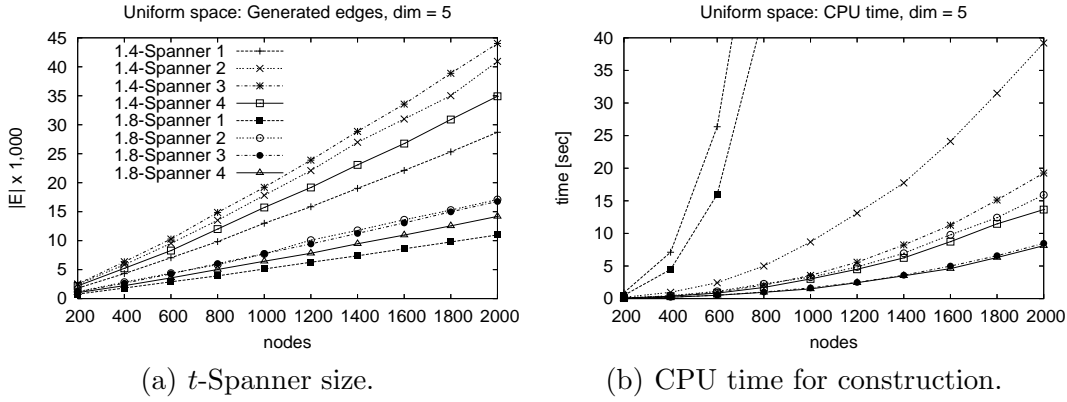


Fig. 13.  $t$ -Spanner construction in a uniform space of dimension  $D = 5$ , for  $t = 1.4$  and  $1.8$ , as a function of the number of nodes. On the left, edges generated ( $t$ -spanner quality). On the right, construction time. In (b), for  $n = 2,000$ ,  $t$ -Spanner 1 reaches 1250 and 830 seconds for  $t = 1.4$  and  $1.8$ , respectively. (b) uses the legend of (a).

	Edges	CPU time
Basic optimized	$1.023e^{0.230D/t^{0.8}} n^{1+\frac{0.101}{t-1}}$	$0.016e^{0.163D/t^{1.3}} n^{3.147+\frac{0.0392}{t-1}}$
Massive edge-insertion	$2.742e^{0.335D/t^{1.8}} n^{1+\frac{0.0693}{t-1}}$	$1.004e^{0.413D/t^{1.8}} n^{2+\frac{0.0723}{t-1}}$
Incremental	$1.983e^{0.308D/t^{1.4}} n^{1+\frac{0.0835}{t-1}}$	$0.560e^{0.182D/t^{1.2}} n^{2+\frac{0.0885}{t-1}}$
Recursive	$1.670e^{0.304D/t^{1.3}} n^{1+\frac{0.0768}{t-1}}$	$0.657e^{0.229D/t^{1.3}} n^{2+\frac{0.0636}{t-1}}$

Table 2

Empirical complexities of our construction algorithms, as a function of  $D$ ,  $n$  and  $t$ . Time is measured in microseconds.

Table 2 shows the least squares fittings on the data using the model  $|E| = a \cdot e^{cD/t^\alpha} \cdot n^{1+\frac{b}{t-1}}$  and  $time = a \cdot e^{cD/t^\beta} \cdot n^{2+\frac{b}{t-1}}$  microseconds. We chose the edge model according to the analytical results of [1,2], with a slight correction to take into account the effect of the dimensionality. As it can be seen,  $t$ -spanner sizes are slightly super-linear for all our construction algorithms, and times are slightly super-quadratic for  $t$ -Spanner 2, 3 and 4. This shows that our algorithms are very competitive in practice. Remember that constructing minimal  $t$ -spanners is NP-complete [33], whereas our algorithms build small  $t$ -spanners in polynomial time.

The analytical results of [1,2] show that the size of a  $t$ -spanner built over a general graph of  $n$  nodes is  $n^{1+O(\frac{1}{t-1})}$ . The terms  $e^{cD/t^\alpha}$  and  $e^{cD/t^\beta}$  represent the usual exponential dependence on the dimension of the metric space (which is also usually hidden in the constant of the big- $O$  notation). We correct the dimensional effect with  $t$ , because as the value of  $t$  increases the effect of the dimensionality diminishes. The CPU time model comes from running  $n$  times a  $\Theta(m \log n)$  CPU time Dijkstra's algorithm, where we neglect the term  $\log n$  to simplify the analysis of the time models. Note that in the case of  $t$ -Spanner

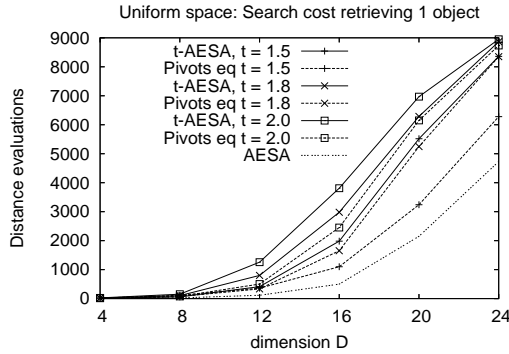


Fig. 14. Range queries in uniform spaces varying  $D \in [4, 24]$ . The data set is composed by 10,000 vectors. We retrieve 1 object per query on average.

1 we modify the time model to  $time = a \cdot e^{cD/t^\beta} \cdot n^{d+\frac{b}{t-1}}$  microseconds, since in this case there is a sub-quadratic shortest path updating algorithm run over matrix  $estim$  for each edge (this time is called  $k^2$  in Table 1).

After this analysis, we select the recursive algorithm to index the metric database, because it yields the best trade-off between CPU time and size of the generated  $t$ -spanner. This result is also verified in Gaussian, string and document spaces (see Sections 5.2.1, 5.3.1 and 5.4.1).

### 5.1.3 Searching

Fig. 14 shows search results on a set formed by 10,000 uniformly distributed vectors indexed with  $t$ -Spanner 4. The 1.5-spanner indexes the database using from 0.17% of the total matrix in dimension  $D = 4$ , to 15.8% in dimension  $D = 24$ , whereas the 2.0-spanner indexes it using 0.08% for  $D = 4$  to 2.02% for  $D = 24$ . With respect to search time, as long as the value of  $t$  diminishes, the performance of  $t$ -AESA improves.

On the other hand, the equivalent pivot-based algorithm (using the same amount of memory to index the space than the  $t$ -spanner) has better performance than  $t$ -AESA. The reasons are analyzed next.

### 5.1.4 Discussion

To explain the poor results of our search algorithm on uniform spaces we need to consider several factors. The first is that the stretch factor  $t$  reduces the discrimination power of our algorithm, as we can only discard objects beyond the extended upper bound of the exclusion range. The second is that, in uniform metric spaces  $([-1, 1]^D, L_p)$ , distances among objects become similar as long as  $D$  increases: the variance of distances is  $\Theta(D^{2/p-1})$  [43], which is constant in Euclidean case. The third is that, in  $([-1, 1]^D, L_2)$ , the average

distance between two objects is  $\Theta(\sqrt{D})$ , thus the distance from a random query to its nearest neighbor is also  $\Theta(\sqrt{D})$  on average (for constant  $n$ ). Therefore, as long as  $D$  increases the query radius also does it, making difficult to discard objects. As a matter of fact, in  $D = 4$  we use radius  $r = 0.137$  to retrieve 1 vector on average, whereas in  $D = 24$  we must use radius  $r = 2.135$ . All these consequences of the curse of dimensionality affect the pivot-based algorithm as well, but our stretch factor  $t$  makes our structure more vulnerable to them.

Nevertheless, this situation will be reverted in real-world metric spaces, where our  $t$ -AESA algorithm beats the pivot-based one and it is very competitive with AESA. We conjecture that this is basically due to the existence of *object clusters*, which naturally occur in real-world metric spaces. Note that in a real-world metric space  $\mathcal{M}$ , it is likely that a real-world query will fall within a cluster, thus meaningful query radii will be very small compared to the average distance in  $\mathcal{M}$ , compensating the loss of discrimination power. We also have to consider that in  $t$ -AESA every object can be used as a pivot; and that the pivot selection criterion (Eq. (7)) tends to select pivots close to the query. Then, we can expect much better results in real-world spaces than in synthetic ones.

In Section 5.2 we experimentally verify our conjecture. For this sake, we model a real-world metric space as a synthetic metric space composed by Gaussian-distributed vectors under Euclidean distance, and then we analyze the behavior of our search algorithm. Later, in Sections 5.3 and 5.4 we show results on real-world metric spaces, namely the space of strings under the edit distance and the space of documents under the cosine distance.

## 5.2 Gaussian-distributed Vectors under Euclidean Distance

Real-life metric spaces have regions called *clusters*, that is, compact zones of the space where similar objects accumulate. With the Gaussian vector space we attempt to simulate a real-world space. The data set is formed by points in a  $D$ -dimensional space with Gaussian distribution forming 256 clusters randomly centered in the range  $[-1, 1]^D$ , for  $D = 20, 40$  and  $80$ . The generator of Gaussian vectors was obtained from [24]. We consider three different standard deviations to make more crisp or more fuzzy clusters ( $\sigma = 0.1, 0.3$  and  $0.5$ ), and  $t \in [1.4, 2.0]$ . Of course, we have not used the fact that the space has coordinates, rather we have treated the points as abstract objects in an unknown metric space.

Computing a single distance takes 1.281, 1.957 and 3.163 microseconds in our machine, for  $D = 20, 40$  and  $80$ , respectively. We experimentally estimate

(according to Section 5.1.1) that the intrinsic dimensions for  $\sigma = 0.1, 0.3$  and  $0.5$  are, respectively, 4, 13 and 18 in the 20-dimensional Gaussian space; 7, 16 and 32 in the 40-dimensional Gaussian space; and 8, 29 and 55 in the 80-dimensional Gaussian space.

In the construction experiments, we only use Gaussian data sets with clusters randomly centered in  $[-1, 1]^{20}$  of varying size  $n \in [200 \dots 2,000]$ . In the search experiments, we index Gaussian datasets formed by 10,000 objects distributed in 256 clusters randomly centered in the range  $[-1, 1]^D$ , for  $D = 20, 40$  and  $80$ , and select 100 random queries not included in the index, using search radii that on average retrieve 1 and 10 objects. In the 20-dimensional Gaussian space we use  $r = 0.442$  and  $0.563$  for  $\sigma = 0.1$ ;  $r = 1.325$  and  $1.690$  for  $\sigma = 0.3$ ;  $r = 2.140$  and  $2.560$  for  $\sigma = 0.5$ . In the 40-dimensional Gaussian space we use  $r = 0.700$  and  $0.818$  for  $\sigma = 0.1$ ;  $r = 2.101$  and  $2.455$  for  $\sigma = 0.3$ ;  $r = 3.501$  and  $4.081$  for  $\sigma = 0.5$ . In the 80-dimensional Gaussian space we use  $r = 1.070$  and  $1.196$  for  $\sigma = 0.1$ ;  $r = 3.210$  and  $3.589$  for  $\sigma = 0.3$ ;  $r = 5.350$  and  $5.981$  for  $\sigma = 0.5$ .

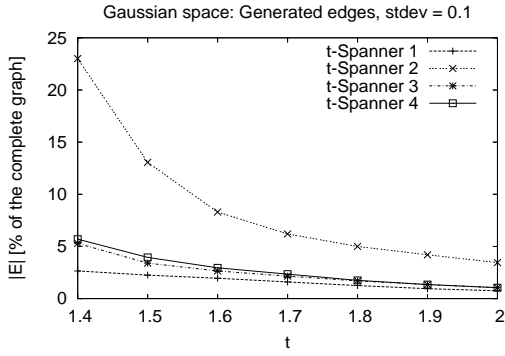
We first compare the construction algorithms, in order to choose the best. Later, search experiments are carried out over the  $t$ -spanner produced by the best construction algorithm. In particular, we aim at empirically verifying the conjecture that  $t$ -spanners profit from clusters more than alternative structures.

### 5.2.1 Construction

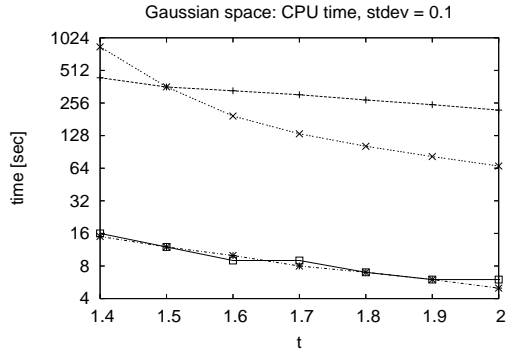
Figs. 15 and 16 compare the four algorithms on the 20-dimensional Gaussian dataset, where we vary the stretch  $t \in [1.4, 2.0]$  and the number of nodes  $n \in [200 \dots 2,000]$ , respectively, with  $\sigma = 0.1, 0.3$ , and  $0.5$ . As it can be seen, all the algorithms produce  $t$ -spanners of about the same quality, although the optimized basic algorithm is consistently better, as expected. It is interesting to note that in the case of  $\sigma = 0.1$ ,  $t$ -Spanner 2 yields the worst quality  $t$ -spanners. This is because, in its first stage, the algorithm inserts a lot of intra-cluster edges and then it tries to connect both inner and peripheral objects among the clusters. Since we need to connect just the peripheral objects, there are many redundant edges that do not improve other distance estimations in the resulting  $t$ -spanner.

In construction time, on the other hand, there are large differences.  $t$ -Spanner 1 is impractically costly, as expected. Also,  $t$ -Spanner 2 is still quite costly in comparison with  $t$ -Spanner 3 and 4.

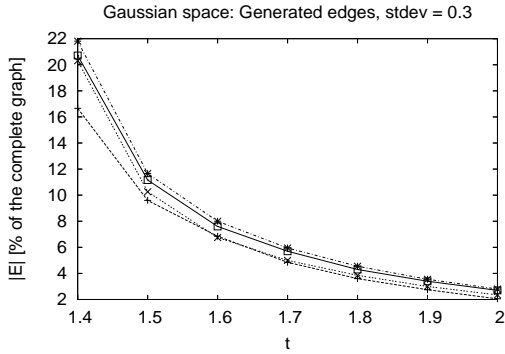
The bad performance of  $t$ -Spanner 2, unlike all the others, *improves* instead of degrading as clusters become more fuzzy, see Figs. 16(b) and 16(d). Nevertheless, the high intrinsic dimensionality of the Gaussian space with  $\sigma = 0.5$



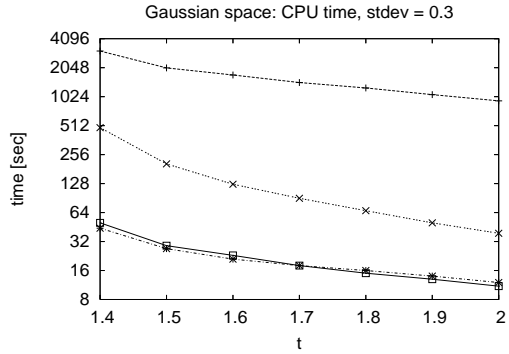
(a)  $t$ -Spanner size for  $\sigma = 0.1$ .



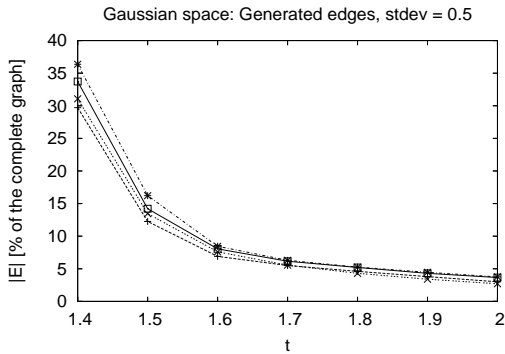
(b) CPU time in construction for  $\sigma = 0.1$ .



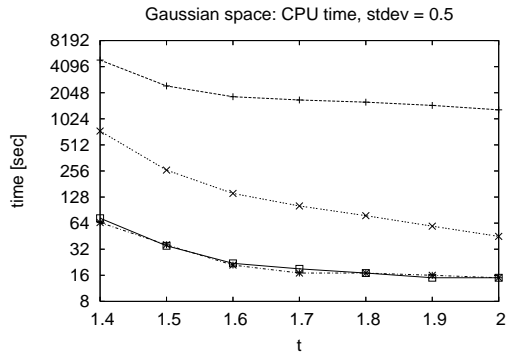
(c)  $t$ -Spanner size for  $\sigma = 0.3$ .



(d) CPU time in construction for  $\sigma = 0.3$ .



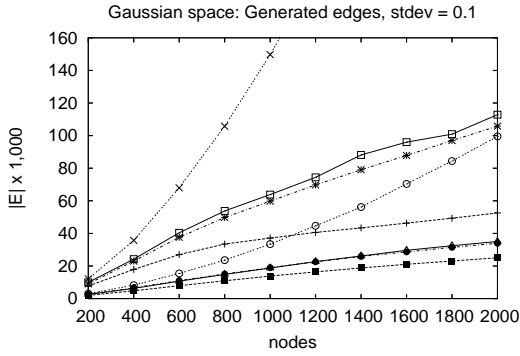
(e)  $t$ -Spanner size for  $\sigma = 0.5$ .



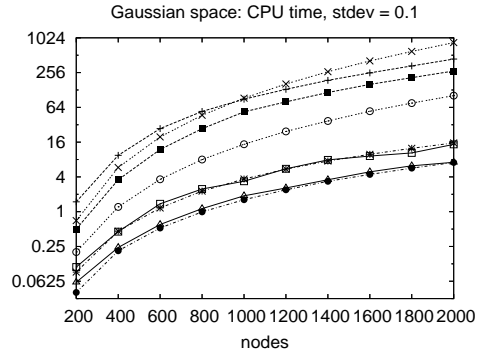
(f) CPU time in construction for  $\sigma = 0.5$ .

Fig. 15.  $t$ -Spanner construction in the Gaussian space of 2,000 nodes with 256 clusters distributed in the range  $[-1, 1]^{20}$  as a function of  $t$ . On the left, edges generated ( $t$ -spanner quality) as a percentage of the complete graph. On the right, construction time, note the log-scale. Each row corresponds to a different variance. All the plots use the legend of (a).

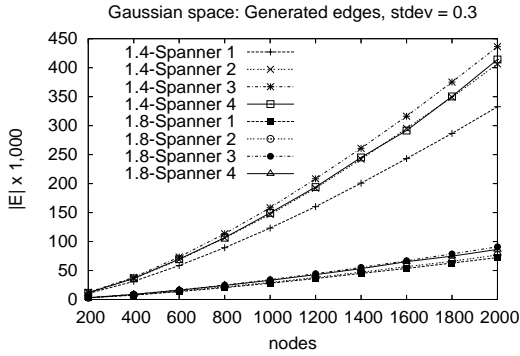
negatively impacts its performance, raising again the CPU time, see Figs. 16(d) and 16(f). Furthermore, the quality of the  $t$ -spanner produced by  $t$ -Spanner 2 also varies from (by far) the worst  $t$ -spanner on crisp clusters to the second best on fuzzy clusters. This is because, on one hand, there are less redundant edges among clusters, and on the other hand, on a uniform space  $t$ -Spanner 2 inserts “better” edges since they come from MSTs (which use the shortest



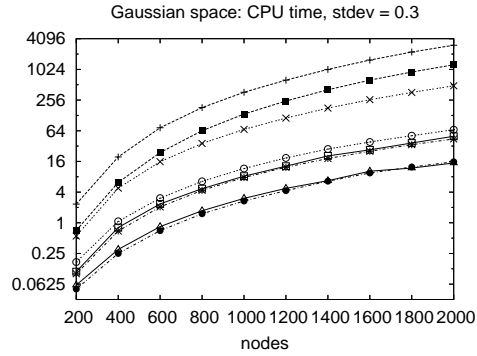
(a)  $t$ -Spanner size for  $\sigma = 0.1$ .



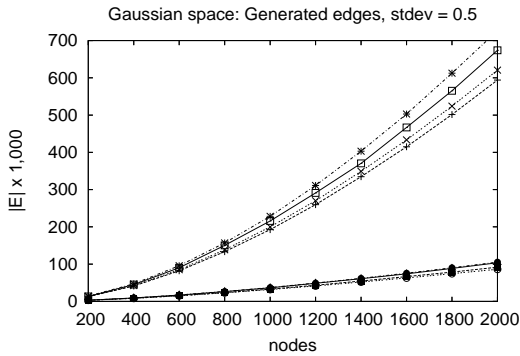
(b) CPU time in construction for  $\sigma = 0.1$ .



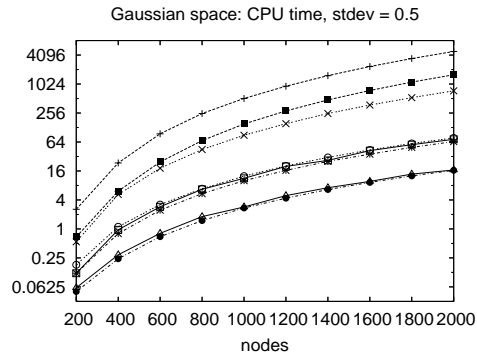
(c)  $t$ -Spanner size for  $\sigma = 0.3$ .



(d) CPU time in construction for  $\sigma = 0.3$ .



(e)  $t$ -Spanner size for  $\sigma = 0.5$ .



(f) CPU time in construction for  $\sigma = 0.5$ .

Fig. 16.  $t$ -Spanner construction in the Gaussian space with 256 clusters distributed in the range  $[-1, 1]^{20}$  as a function of the number of nodes. On the left, edges generated ( $t$ -spanner quality). On the right, construction time, note the log-scale. Each row corresponds to a different variance. In (a), 1.4-Spanner 2 reaches 460,000 edges. In (e), 1.4-Spanner 3 reaches 727,000 edges. All the plots use the legend of (c).

possible edges).

Figs. 15 and 16 show that, the lower  $\sigma$ , the lower the number of edges and CPU time. This is because the edge selection mechanisms of our algorithms profit from the clustered structure. In practice,  $t$ -Spanner 1, 3 and 4 can make good

distance approximations between the clusters by using few edges. Thus, the generated  $t$ -spanners adapt well to, and benefit from, the existence of clusters.

The incremental and recursive algorithms are quite close in both measures, being by far the fastest algorithms. The recursive algorithm usually produces slightly better  $t$ -spanners thanks to its more global edge analysis. Note that, for  $t$  as low as 1.5, we obtain  $t$ -spanners whose size is 5% to 15% of the full graph.

It is interesting to notice that, for fuzzy clusters, there is a sharp increase in construction time and  $t$ -spanner size when we move from  $t = 1.5$  to  $t = 1.4$ . The effect shows up for smaller values of  $t$  on crisper clusters. A possible explanation is that, for large enough  $t$ , a single edge from a cluster to another is enough to obtain a  $t$ -spanner over both clusters. Thus, few inter-cluster edges are necessary to complete the  $t$ -spanner. However, when  $t$  is reduced below some threshold, the size of the edge set suddenly explodes as we need to add many inter-cluster edges to fulfill the  $t$ -condition from one cluster to each other.

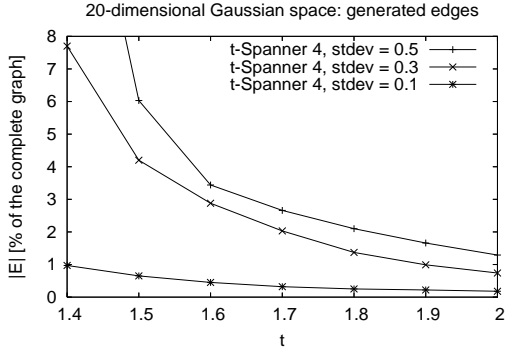
We show in Table 3 our least squares fittings on the data using the model  $|E| = a \cdot n^{1+\frac{b}{t-1}}$  and  $time = a \cdot n^{2+\frac{b}{t-1}}$  microseconds for  $t$ -Spanner 2, 3 and 4. Once again, we modify the  $t$ -Spanner 1 time model to  $time = a \cdot n^{c+\frac{b}{t-1}}$  microseconds so as to consider the sub-quadratic updating of matrix *estim*. In this metric space the effect of  $\sigma$  is absorbed by the constants. This model has also been chosen according to the analytical results of [1,2]. As it can be seen, we obtain the same conclusions than in the previous section, that is,  $t$ -spanner sizes are slightly super-linear and times are slightly super-quadratic. This confirms that our construction algorithms are very competitive in practice.

This analysis also confirms the selection of the recursive algorithm to index the metric database, as it yields the best trade-off between CPU time and size of the generated  $t$ -spanner.

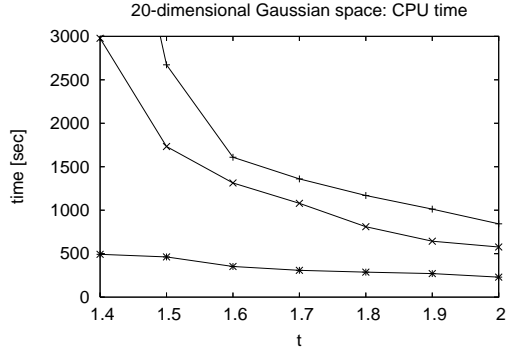
### 5.2.2 Searching

For this section we will use the datasets of 10,000 Gaussian vectors in 20, 40, and 80 dimensions. Fig. 17 shows the construction results when indexing these datasets using  $t$ -Spanner 4 varying  $t \in [1.4, 2]$ . For  $\sigma = 0.1, 0.3$  and  $0.5$ , the 1.4-spanner indexes the database using 0.97%, 7.70% and 16.32% of the memory required by AESA, respectively, in dimension 20. For higher dimensions, only  $t = 2$  is practical.

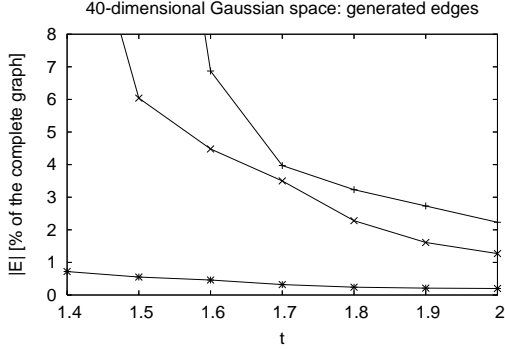
Fig. 18 shows search results in dimension 20. It can be seen that the performance of  $t$ -AESA improves as the value of  $t$  decreases, both to retrieve 1 and



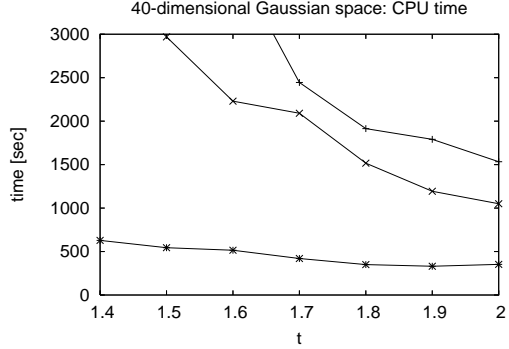
(a) Edges generated in 20 dimensions.



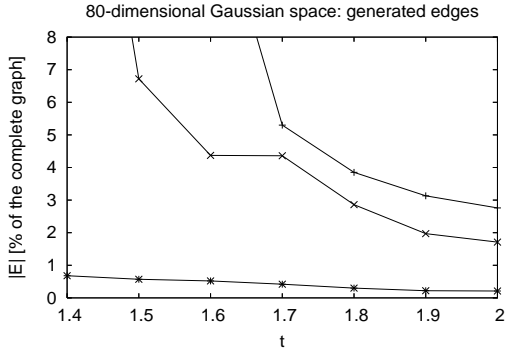
(b) CPU time in 20 dimensions.



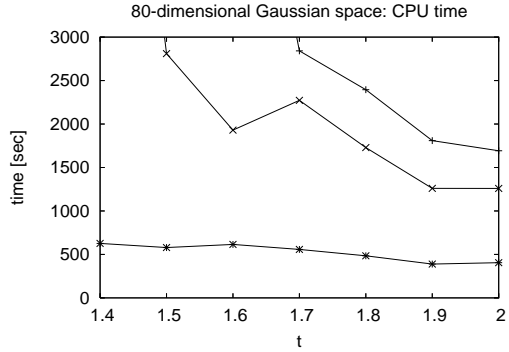
(c) Edges generated in 40 dimensions.



(d) CPU time in 40 dimensions.



(e) Edges generated in 80 dimensions.



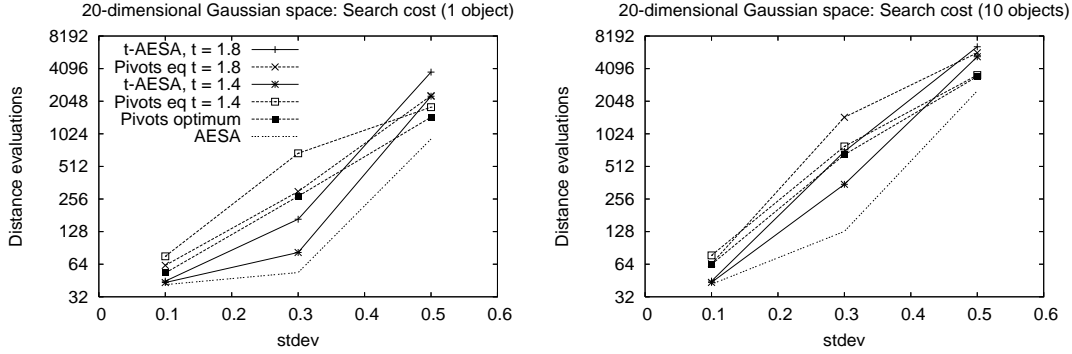
(f) CPU time in 80 dimensions.

Fig. 17.  $t$ -Spanner construction in a Gaussian space of 10,000 nodes with 256 clusters distributed in the range  $[-1, 1]^D$ , for  $\sigma = 0.1, 0.3$  and  $0.5$ , using  $t$ -Spanner 4 as a function of  $t$ . On the left, edges generated ( $t$ -spanner quality) as a percentage of the complete graph. On the right, construction time. Each row corresponds to a different  $D$ . In  $D = 20$ , for  $\sigma = 0.5$ ,  $t$ -Spanner 4 reaches 16.3% (a) and 6,200 seconds (b). In  $D = 40$ , for  $\sigma = 0.3$ , it reaches 13.9% (c) and 6,750 seconds (d); and for  $\sigma = 0.5$ , it reaches 55.8% (c) and 19,800 seconds (d). Finally, in  $D = 80$ , for  $\sigma = 0.3$ , it reaches 20.0% (e) and 9,280 seconds (f); and for  $\sigma = 0.5$ , it reaches 89.4% (e) and 23,600 seconds (f). All the plots use the legend of (a).

	Basic optimized	Massive edge insertion	Incremental	Recursive
$\sigma = 0.1$				
CPU time	$1.16 n^{2.44+\frac{0.07}{t-1}}$	$1.67 n^{2+\frac{0.24}{t-1}}$	$0.670 n^{2+\frac{0.10}{t-1}}$	$0.909 n^{2+\frac{0.08}{t-1}}$
Edges	$5.76 n^{1+\frac{0.10}{t-1}}$	$6.50 n^{1+\frac{0.18}{t-1}}$	$6.17 n^{1+\frac{0.13}{t-1}}$	$5.77 n^{1+\frac{0.14}{t-1}}$
$\sigma = 0.3$				
CPU time	$0.054 n^{2.99+\frac{0.11}{t-1}}$	$1.52 n^{2+\frac{0.22}{t-1}}$	$0.771 n^{2+\frac{0.13}{t-1}}$	$0.865 n^{2+\frac{0.13}{t-1}}$
Edges	$5.69 n^{1+\frac{0.18}{t-1}}$	$5.41 n^{1+\frac{0.19}{t-1}}$	$6.52 n^{1+\frac{0.19}{t-1}}$	$6.50 n^{1+\frac{0.18}{t-1}}$
$\sigma = 0.5$				
CPU time	$0.027 n^{3.08+\frac{0.13}{t-1}}$	$1.33 n^{2+\frac{0.25}{t-1}}$	$0.587 n^{2+\frac{0.17}{t-1}}$	$0.650 n^{2+\frac{0.17}{t-1}}$
Edges	$4.89 n^{1+\frac{0.21}{t-1}}$	$4.50 n^{1+\frac{0.22}{t-1}}$	$5.20 n^{1+\frac{0.22}{t-1}}$	$5.37 n^{1+\frac{0.21}{t-1}}$

Table 3

Empirical complexities of our construction algorithms, as a function of  $n$  and  $t$ . Time is measured in microseconds.



(a) Search cost to retrieve 1 object.

(b) Search cost to retrieve 10 objects.

Fig. 18. Range queries in a Gaussian space of 10,000 nodes with 256 clusters distributed in the range  $[-1, 1]^{20}$ , for  $\sigma = 0.1, 0.3$  and  $0.5$ . On the left, retrieving 1 object. On the right, retrieving 10 objects. (b) uses the legend of (a). Note the log-scales in the plots.

10 objects. We also show the pivot-based algorithm performance when using the optimum number of pivots (found by hand).

On the other hand, the cluster diameter also influences the performance of  $t$ -AESA. On crisp clusters, the results are competitive against AESA and better than the pivot-based algorithm, since the  $t$ -spanner adapts to and benefits from the existence of clusters. For instance, with  $\sigma = 0.1$ , 1.4-AESA retrieves 1 and 10 objects using 1.05 and 1.04 times the distance evaluations of AESA, respectively, whereas the pivot-based algorithm uses 1.51 and 1.54 times the number of evaluations of AESA, respectively. Moreover, the pivot-based al-

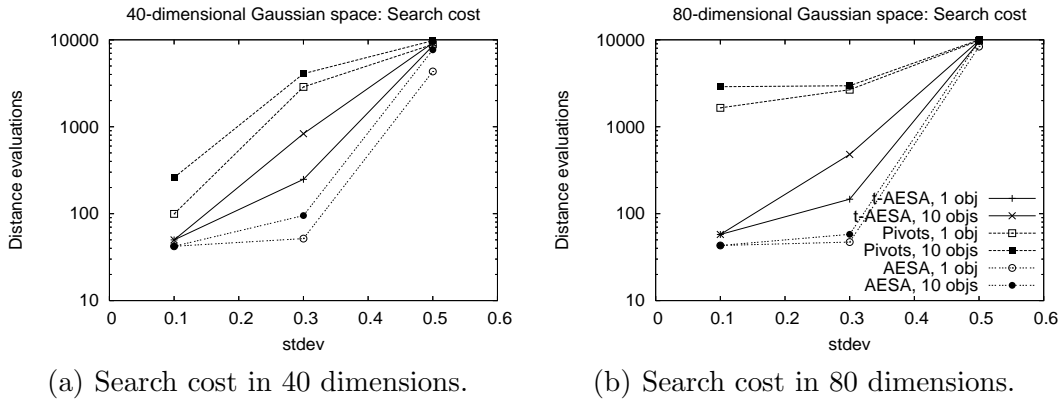


Fig. 19. Range queries in a Gaussian space of 10,000 nodes with 256 clusters for  $\sigma = 0.1, 0.3$  and  $0.5$ . We index the space with a 2.0-spanner. In (a), clusters distribute in the range  $[-1, 1]^{40}$ . In (b), clusters distribute in the range  $[-1, 1]^{80}$ . (a) uses the legend of (b). Note the log-scales in the plots.

gorithm using the optimum number of pivots takes 1.28 and 1.52 times the number of evaluations of AESA, respectively. With  $\sigma = 0.3$ ,  $t$ -AESA is still better than the pivot-based algorithm. As a matter of fact, 1.4-AESA retrieves 1 and 10 objects by using 1.53 and 2.72 times the distance evaluation of AESA, and the optimum pivot-based algorithm uses 5.07 and 5.11 times the distance evaluations of AESA for 1 and 10 objects. The situation is reverted in fuzzy clusters, as expected from Section 5.1.3, where the objects are distributed almost uniformly and the  $t$ -spanner notoriously loses discrimination power.

Fig. 19 shows the results on higher dimensions and  $t = 2$ . For crisp clusters ( $\sigma = 0.1$ )  $t$ -AESA needs few more distance computations than AESA, both to retrieve 1 or 10 objects. For instance, Fig. 19(a)/(b) shows that in 40/80 dimensions  $t$ -AESA uses 1.19/1.34 times the number of evaluations of AESA, both to retrieve 1 and 10 objects. With  $\sigma = 0.3$   $t$ -AESA is better, by far, than the pivot-based technique. Finally, with  $\sigma = 0.5$  all the techniques dramatically fall down in performance, as with that  $\sigma$  the space is almost uniformly distributed, and this makes up an intractable scenario on high dimensions.

Hence we experimentally verify the conjecture that  $t$ -spanners take advantage of the clusters, which occur naturally in real-world metric spaces. The crisper the clusters, the better the performance of  $t$ -spanner techniques. It is known that all search algorithms improve in the presence of clusters. However,  $t$ -spanner based algorithms improve more than, for example, pivot-based algorithms.

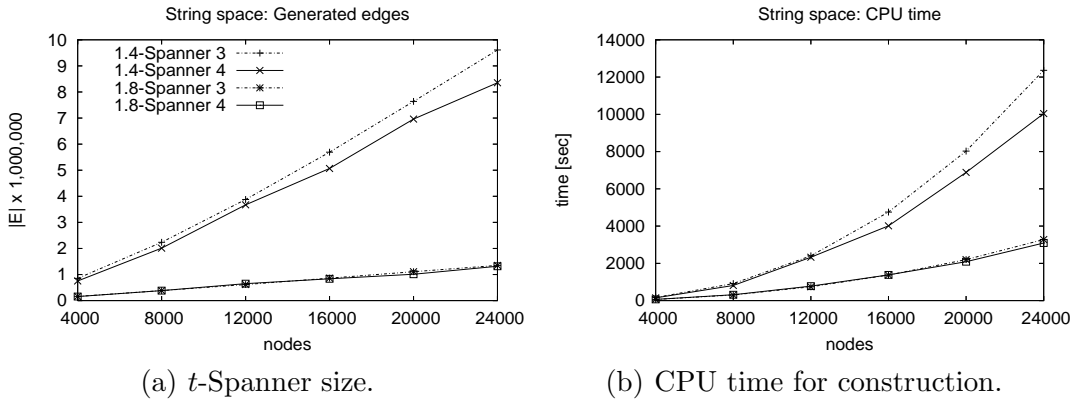


Fig. 20.  $t$ -Spanner construction on the space of strings, for increasing  $n$ . On the left, number of edges generated ( $t$ -spanner quality). On the right, construction time. (b) uses the legend of (a).

### 5.3 Strings under Edit Distance

The string metric space under the edit distance has no coordinates. The edit distance is a discrete function that, given two strings, measures the minimum number of character insertions, deletions and substitutions needed to transform one string to the other [32]. Our database is an English dictionary, where we index a subset of  $n = 24,000$  randomly chosen words. On average, a distance computation takes 1.632 microseconds.

Since these tests are more massive, we leave out the optimized basic and the massive edge-insertion algorithms in the construction experiments, as they were too slow. Anyway, we performed a test with a reduced dictionary in order to validate the decision of leaving them out, and to experimentally estimate the intrinsic dimensionality of the string space as  $dim = 8$ .

In the search experiments, we select 100 queries at random from dictionary words not included in the index. We search with radii  $r = 1, 2, 3$ , which return on average 0.0041%, 0.036% and 0.29% of the database, that is, approximately 1, 8 and 66 words of the english dictionary, respectively.

#### 5.3.1 Construction

Fig. 20 shows that, also for strings, the number of edges generated is slightly super-linear ( $8.03 n^{1+\frac{0.16}{t-1}}$  for  $t$ -Spanner 3 and  $8.45 n^{1+\frac{0.15}{t-1}}$  for  $t$ -Spanner 4), and the construction time is slightly super-quadratic ( $1.46 n^{2+\frac{0.10}{t-1}}$  microseconds for  $t$ -Spanner 3 and  $1.67 n^{2+\frac{0.09}{t-1}}$  for  $t$ -Spanner 4). The recursive algorithm is almost always a bit better than the incremental algorithm in both aspects.

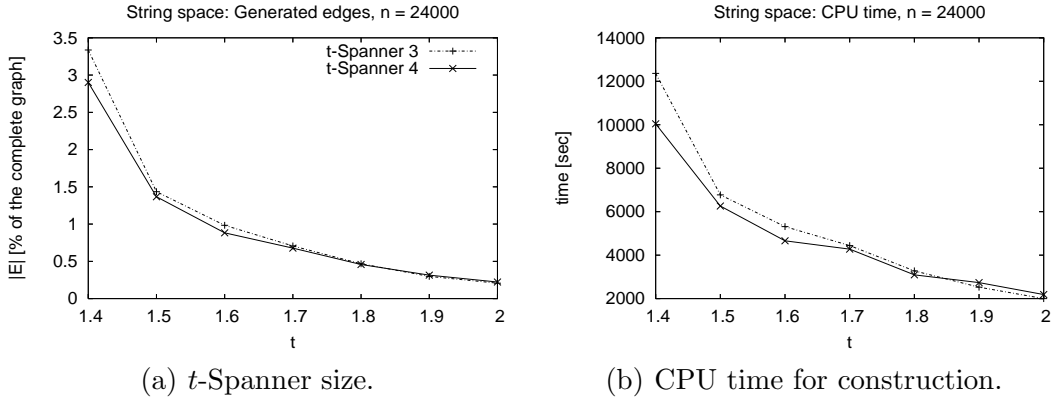


Fig. 21.  $t$ -Spanner construction on the space of strings as a function of  $t$ , for  $n = 24000$ . On the left, number of edges generated ( $t$ -spanner quality) as a percentage of the complete graph. On the right, construction time. (b) uses the legend of (a).

Fig. 21 shows construction results when indexing the whole subset of 24000 strings varying  $t$ . The full graph of 24,000 nodes has 288 million edges, whereas a 1.4-spanner has only 8.35 million edges (2.90% of the complete graph). Once again, Figs. 21(a) and 21(b) confirm the selection of  $t$ -Spanner 4.

### 5.3.2 Searching

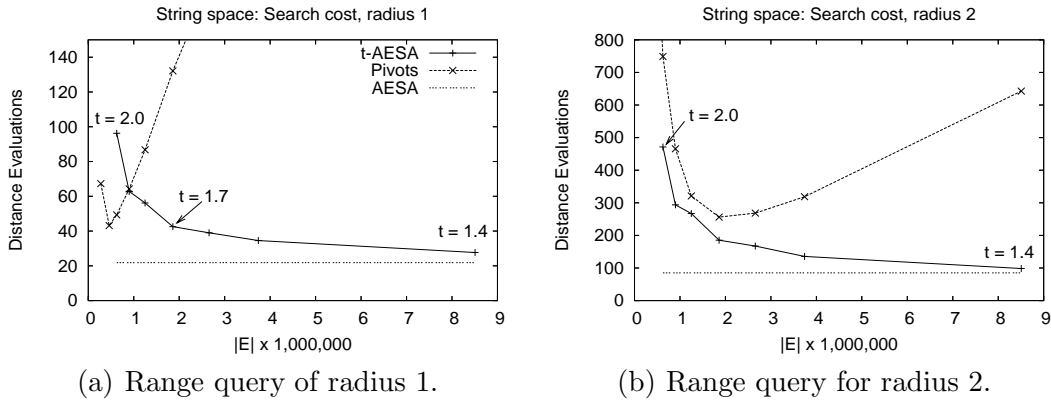
Fig. 22 presents the results as a space/time plot. We have chosen to draw a line to represent AESA although, since it permits no space-time trade-offs, a point would be the correct representation. The position of this point in the  $x$  axis would be  $288 \times 10^6$ , about 1.7 yards from the right end of the plot.

Note that, while pivot-based algorithms have a limit where giving them more memory does not improve their performance,  $t$ -AESA always improves with more memory.

Fig. 22(a) shows that, with radius 1, the pivot-based algorithm has better performance than  $t$ -AESA with  $t = 2.0$ . Furthermore, the equivalent pivot-based algorithm performs better than  $t$ -AESA for  $t > 1.7$ , and it uses less memory. Only with values of  $t \leq 1.7$  (which produce an index using more memory),  $t$ -AESA works systematically better, becoming very competitive for  $t = 1.4$ , where it makes just 1.27 times the distance evaluations of AESA.

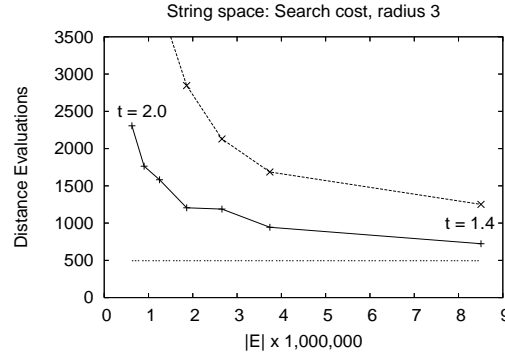
Figs. 22(b) and 22(c) show that with radii 2 and 3,  $t$ -AESA has better performance than pivots for all  $t \in [1.4, 2.0]$ . For example, 1.4-AESA uses 1.16 and 1.46 times the distance evaluations of AESA with radii 2 and 3, respectively.

In order to understand these so favorable results, we have to take into account that, in addition to the fact that the string space has small radius clusters,



(a) Range query of radius 1.

(b) Range query for radius 2.



(c) Range query for radius 3.

Fig. 22. Distance evaluations on the string space. In (a), search radius  $r = 1$ ; the pivot-based algorithm reaches 540 distance evaluations for the pivot-index equivalent to a 1.4-spanner (8.5 million edges). In (b), search radius  $r = 2$ . In (c), search radius  $r = 3$ . All the plots use legend of (a).

the edit distance is a discrete function. This means that the approximated distances are rounded down to the closest integer. This fact can be interpreted as if the effective  $t'$  of the structure were lower than the nominal  $t$ , so that the discrimination power lost is less than in continuous metric spaces. Overall, this improves the discrimination power of  $t$ -AESA.

#### 5.4 Documents under Cosine Distance

In Information Retrieval, documents are usually represented as unitary vectors of high dimensionality [5]. The idea consists in mapping the document to a vector of real values, so that each dimension is a vocabulary word and the relevance of the word to the document (computed using some formula) is the coordinate of the document along that dimension.

With this model, document similarity is assessed through the inner product between the vectors. Note that the inner product between two exactly equal

documents is one, since both documents are represented by the same vector. As long as the documents are more different, the inner product between the vectors representing the documents goes to zero. As we are looking for a distance, we consider the angle between these vectors. This way, the cosine distance is simply the arc cosine of the inner product between the vectors [5], and this satisfies the triangle inequality. Similarity under this model is very expensive to calculate.

In the construction experiments, we use a data set formed by 1,200 documents obtained from TREC-3 collection (<http://trec.nist.gov/>), and exclude the massive edge-insertion algorithm, which was too slow (the reason, this time, is that  $t$ -Spanner 2 is the algorithm that makes, by far, most distance computations), with  $t \in [1.4, 2.0]$ .

In the search experiments, we select 50 queries at random from the documents not included in the index, and search with radii chosen to retrieve, on average, 1 and 10 documents per query ( $r = 0.13281$  and  $0.16659$ , respectively), with  $t \in [1.4, 2.0]$ .

In most of these experiments, we maintained the whole document set in memory in order to avoid the disk time, but in real applications we cannot choose this option, since the data set can be arbitrarily large. We have included an experiment with the documents held on disk in Section 5.4.2 to demonstrate the effect of such an expensive distance function.

#### 5.4.1 Construction

Fig. 23(a) shows that  $t$ -Spanner 1, 3 and 4 produce  $t$ -spanners of about the same quality. However, as shown in Fig. 23(b) the optimized basic algorithm is much more expensive than the other two, which are rather similar. Yet, these time differences are not very large if we compare them to those of Fig. 15. For instance, for  $t = 1.4$ ,  $t$ -Spanner 1 uses 3 times more CPU time than  $t$ -Spanner 3 or 4, whereas in Gaussian or uniform spaces  $t$ -Spanner 1 is 64 times slower. This is because the cost to compute the cosine distance is clearly the dominant term of the CPU time cost, and this is always close to  $n(n - 1)/2$  distances. As a matter of fact, for  $t = 2$  the CPU time of  $t$ -Spanner 1, 3 and 4 are almost equal. Note that the complete graph of 1,200 nodes has about 720 thousand edges, and a 2.0-spanner has only 28.7 thousands (3.99% of the complete graph).

#### 5.4.2 Searching

Fig. 24 shows that  $t$ -AESA can achieve better performance than the pivot-based algorithm, and it is extremely competitive against AESA. For example,

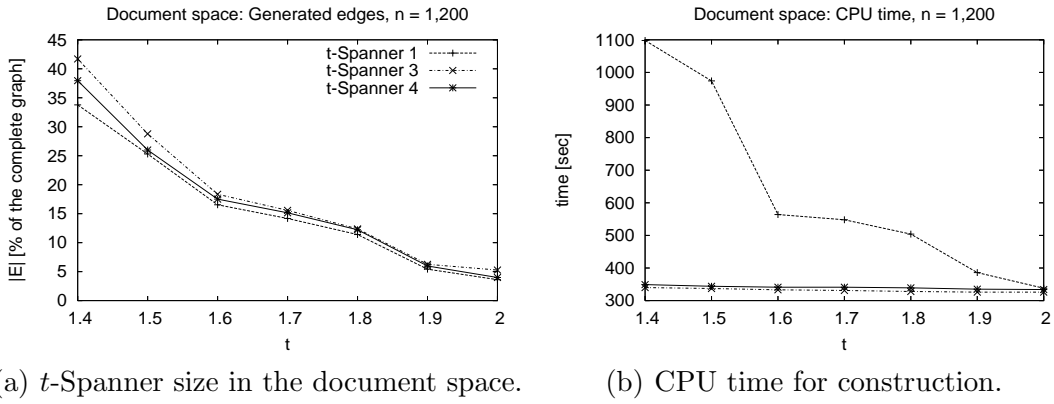


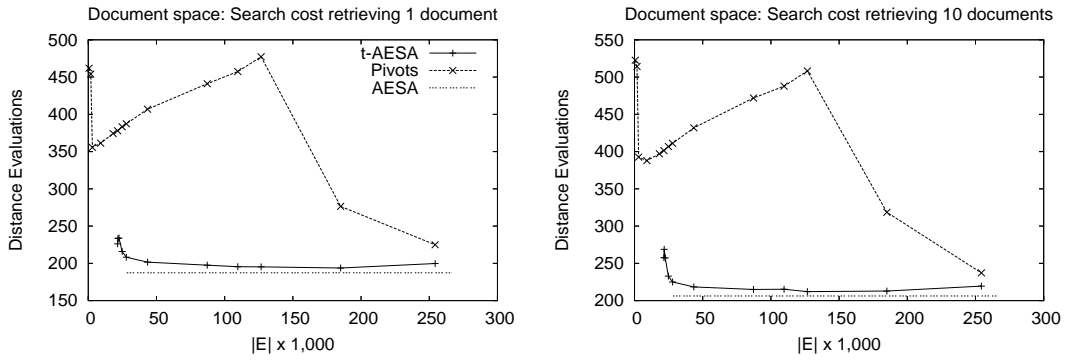
Fig. 23.  $t$ -Spanner construction on the set of documents, as a function of  $t$ . On the left, edges generated ( $t$ -spanner quality) as a percentage of the complete graph. On the right, construction time. (b) uses the legend of (a).

for  $t = 2.0$ ,  $t$ -AESAs makes 1.09 times the distance evaluations of AESA in order to retrieve the most similar document, and 1.08 times to retrieve 10 documents. We have again chosen to draw a line to represent AESA.

When comparing to pivots, we note that pivots do better than  $t$ -AESAs when the amount of memory is very limited (see the smallest values of  $|E| \times 1,000$ ). Yet, it can be seen that pivots are not so efficient to take advantage of more memory when it is available. Actually their initial achievement (350–400 distance computations) are not surpassed until they use much more memory, and at that point they do not beat  $t$ -AESAs either.

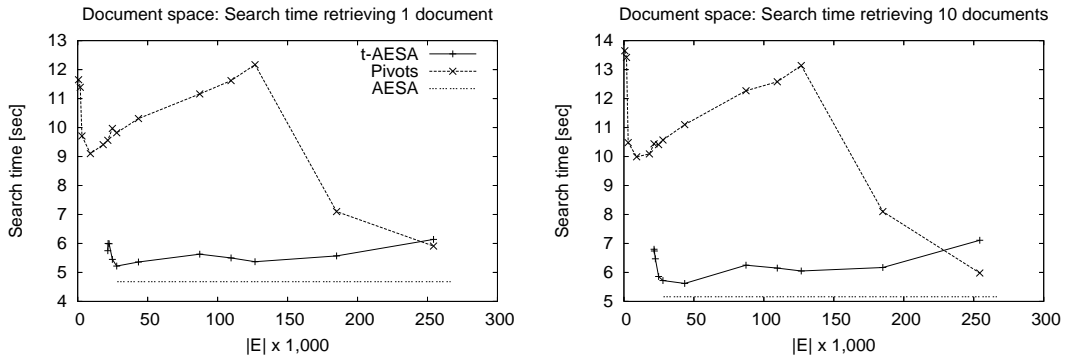
Up to now we have considered only number of distance evaluations at search time, in the understanding that in some metric spaces the distance is expensive enough to absorb the extra CPU time we incur with the graph traversals. There are many metric spaces where this is the case, and the document space is a good example to illustrate this situation. The documents are stored on disk and we measure the overall elapsed time to retrieve 1 and 10 documents.

Fig. 25 shows that, even considering side computations,  $t$ -AESAs can achieve better performance than the pivot-based algorithm. When the overall time is considered,  $t$ -AESAs has an optimum  $t$ -spanner size of around 25,000 edges. This size is rather small and is achieved using  $t = 2.1$ . It is equivalent to using 41 pivots. To achieve a similar result with pivots, one has to spend about 10 times more space.



(a) Search cost to retrieve 1 document. (b) Search cost to retrieve 10 documents.

Fig. 24. Distance evaluations on the document space. On the left, retrieving 1 document. On the right, retrieving 10 documents. (b) uses the legend of (a).



(a) Search time to retrieve 1 document. (b) Search time to retrieve 10 documents.

Fig. 25. Elapsed search time in the document space. On the left, retrieving 1 document. On the right, retrieving 10 documents. (b) uses the legend of (a).

## 6 $t$ -Spanners as Dynamic Structures for Metric Space Searching

In many real applications, one does not know the whole object set at index construction time, but rather objects are added/removed to/from the set along time. Thus, it is necessary to update the index to represent the current status of the data set.

Hence, in order to use the  $t$ -spanner as a metric database representation, we have to grant it the ability of allowing efficient object insertions and deletions without degrading the performance of the object retrieval operations.

We show in this section how to implement a dynamic index for metric spaces based on  $t$ -spanners. These algorithms allow us to efficiently *update* the structure upon insertions and deletions of objects, while maintaining its quality.

### 6.1 Inserting an Object into the $t$ -Spanner

Assume we want to insert an object  $u$  into a  $t$ -spanner  $G'(V, E)$  so as to obtain a new  $t$ -spanner  $G'_u(V \cup \{u\}, E_u)$ . Since we are only interested in that the  $t$ -estimations from  $u$  towards elements of  $V$  fulfill the  $t$ -condition and  $G'(V, E)$  is already a well-formed  $t$ -spanner, this is the perfect situation to use the incremental algorithm (Section 4.4) for  $u$ .

Therefore, constructing a  $t$ -spanner by successive object insertions is equivalent to using the incremental algorithm over the whole object set.

### 6.2 Deleting an Object from the $t$ -Spanner

Elements handled in metric spaces are in many cases very large. Thus, upon an object deletion, it is mandatory to effectively remove it. As a consequence, folklore solutions such as marking objects as eliminated without effectively removing them are not acceptable in the metric space scenario.

We present two choices to remove elements from a  $t$ -spanner: *lazy deletion* and *effective deletion*.

**Lazy deletion** consists in removing the object and leaving its node and edges untouched. Those nodes that do not represent anymore an object are said to be *empty*. Empty nodes are treated as regular nodes at search time, except that they are not considered to be candidate nodes. As a consequence, they are never chosen as pivots nor reported. In Fig. 3, this corresponds to changing just line 1, so that  $\mathcal{C}$  is initially  $\mathbb{U}$  minus the empty nodes.

This choice has the advantage of taking constant time and not modifying the  $t$ -spanner structure. In order to preserve the long-term quality of the graph, periodic global reconstructions should be carried out (see Section 6.3).

**Effective deletion** consists in eliminating not only the object, but also its graph node and its incident edges. In order to preserve the  $t$ -condition, we have to make a local  $t$ -spanner reconstruction around the deleted object by using “temporal” edges. Note that some temporal edges could be unnecessary to preserve the  $t$ -condition, and that for each inserted edge we need to compute one additional distance. At search time, temporal edges are treated as regular edges. Just as for empty nodes in lazy deletion, global periodic reconstructions are necessary to get rid of the temporal edges.

We envision two connection mechanisms to perform the local  $t$ -spanner reconstruction:

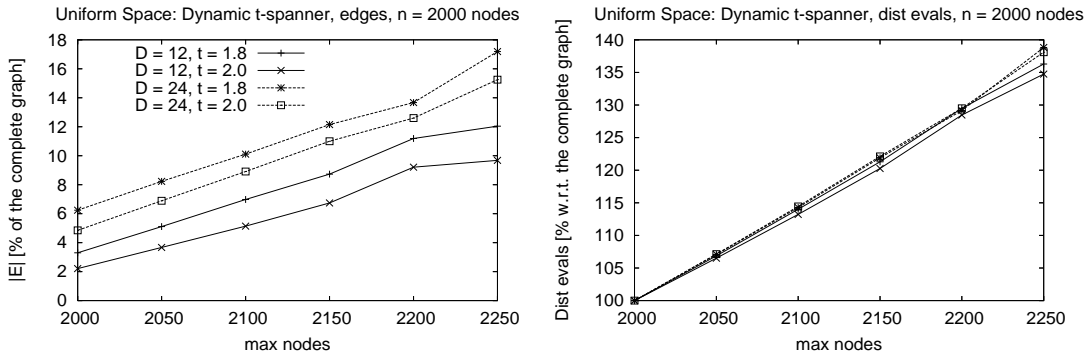
- *Clique connection*: it consists in connecting all the neighbors of the deleted vertex to each other using temporal edges. This mechanism has the advantage of freeing the memory used by the node and its incident edges, but the disadvantage of inserting some unnecessary edges, which degrade the quality of the  $t$ -spanner.
- *Conservative connection*: it consists in connecting neighbors of the deleted vertex, with temporal edges, only if the current distance  $t$ -estimation among them is greater than *before* the vertex deletion. This choice has the advantage of inserting only some edges of the clique, but the disadvantage of taking more CPU time per deletion.

Note that an alternative to conservative connection is *local  $t$ -spanner connection*, where we use temporal edges to connect neighbors of the deleted vertex whose current distance estimation is greater than the  $t$ -distance *allowed* for them. This mechanism takes one additional distance evaluation per checked edge, even if the distance is not inserted in the  $t$ -spanner. This option has the advantage of preserving the  $t$ -spanner quality and taking less CPU time per deletion compared to the previous mechanism. The disadvantage is that this local reconstruction does not necessarily preserve the global  $t$ -spanner property. That is, even if we ensure that the  $t$ -condition is locally restored among neighbors, it might be that some distant nodes get their distance not  $t$ -estimated. These distances have to be detected and patched by adding direct edges. Periodical global reviews of the  $t$ -spanner can be used both to detect these distant edges, and to remove the excess of direct edges in favor of shorter ones. Yet, the method does not guarantee that we actually have a  $t$ -spanner.

### 6.3 Remodeling the $t$ -Spanner

After successive deletions (using either lazy or effective deletion) the quality of the  $t$ -spanner may be degraded. This motivates the need of a “reconstruction” algorithm to be run periodically in order to maintain the structure quality, that is, to maintain a  $t$ -spanner that appropriately models the current state of the metric database.

The remodeling algorithm has two stages. First, depending on the deletion strategy, it deletes either empty nodes and their incident edges, or temporal edges. Second, it builds the  $t$ -spanner by using the recursive algorithm, starting with the nodes and edges that remain after the elimination. This strategy allows us to reuse previous work.



(a) Size of the  $t$ -spanner after updates. (b) Distance evaluations for the whole process.

Fig. 26. Dynamic  $t$ -spanners for  $D = 12$  and  $24$  and  $t = 1.8$  and  $2.0$  as a function of the maximum number of nodes. On the left, edges generated ( $t$ -spanner quality). On the right, distance computations required by the reconstruction. (b) uses the legend of (a).

#### 6.4 Experimental Results with Dynamic $t$ -Spanners

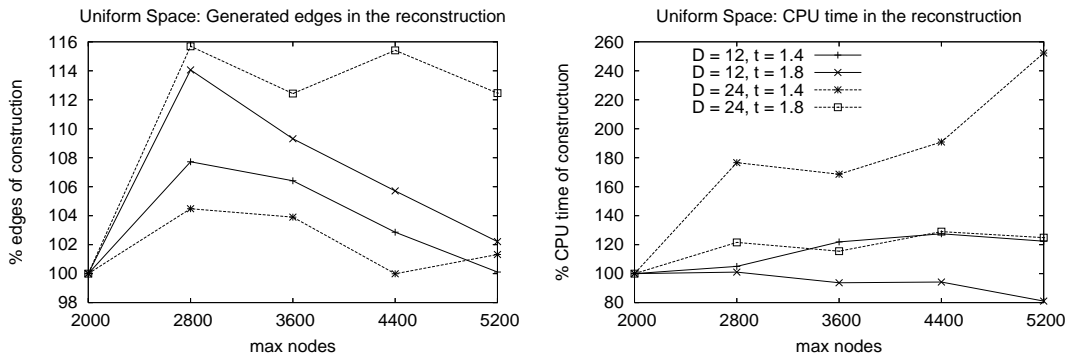
In this section we show that dynamic  $t$ -spanners are a robust technique to index a metric database upon insertions and deletions of objects. We first study effective deletion as an alternative to lazy deletion, and then consider the effect of periodic reconstructions.

##### 6.4.1 Performance of Effective Deletion

While the performance of lazy deletion is immediate in terms of the resulting  $t$ -spanner size and deletion cost, those measures deserve experimental study for effective deletion.

We start with a dataset of  $n$  vectors in  $([-1, 1]^D, L_2)$  that we index with  $t$ -Spanner 4. Then we add nodes until reaching a maximum value  $n_{max} > n$ , by using the incremental algorithm for each new object as suggested in Section 6.1. Now we mark  $n_{max} - n$  objects at random and delete them using the conservative connection technique of Section 6.2.

Fig. 26 shows the results on insertion and deletion over  $([-1, 1]^D, L_2)$ , for  $D \in [4, 24]$ . We start by indexing  $n = 2,000$  vectors, and then vary  $n_{max}$  from 2,050 to 2,250. Fig. 26(a) shows the number of edges when we add objects to the  $t$ -spanner and then remove the same amount at random using the conservative connection technique. Fig. 26(b) shows the number of distance computations required by the whole process of insertions and deletions, compared to just inserting 2,000 nodes.



(a) Size of the reconstructed  $t$ -spanner.

(b) CPU time for reconstruction.

Fig. 27.  $t$ -Spanner reconstruction in uniform spaces for  $D = 12$  and 24, as a function of the maximum number of nodes. On the left, edges generated ( $t$ -spanner quality) measured as a percentage with respect to the original. On the right, reconstruction CPU time measured as a percentage with respect to the original. (a) uses the legend of (b).

It can be seen that the number of edges sharply increases under effective deletion, even after a few updates. In the best case ( $D = 24, t = 1.8$ ) the number of edges after reaching 2,200 elements and then deleting 200 is more than twice that of the initial graph (before any update). If we used lazy deletion, this ratio would be below 1.11. Thus, we pay a high price in terms of edges for having removed the node. In addition, we pay a high price in terms of distance computations (Fig. 26(b)), as opposed to zero in the case of lazy deletion. This shows that lazy deletion is clearly preferable over effective deletion.

#### 6.4.2 Remodeling

We test now the performance of our  $t$ -spanner remodeling technique. We start with a dataset of  $n$  vectors in  $([-1, 1]^D, L_2)$ , insert new nodes until reaching  $n_{max}$ , and then delete random nodes until we are back with  $n$  nodes. Now, we use the remodeling technique from Section 6.3. Note that the deletion strategy we use is irrelevant for this experiment.

Fig. 27 shows the results for  $D \in [4, 24]$ . We start by indexing  $n = 2,000$  vectors, and then vary  $n_{max}$  from 2,800 to 5,200. In Fig. 27(a) we show the ratio between edge set size after the reconstruction and that of the initial indexing. Analogously, in Fig. 27(b) we show the ratio between the reconstruction time and initial construction time.

It can be seen that reconstruction is quite effective, maintaining the  $t$ -spanner quality relatively similar to its original version even after many updates take place. This shows that the combination of lazy deletion and periodic remodeling is a robust alternative for dynamic  $t$ -spanners (note that in this case the  $t$ -spanner quality and search performance are unaltered, yet objects are not

physically removed until the next remodeling takes place).

## 7 Conclusions and Further Work

We have presented a new approach to metric space searching, which is based on using a  $t$ -spanner data structure as an approximate map of the space. This permits us trading space for query time.

We started by proposing an algorithm to solve range queries over the  $t$ -spanner. It is based on simulating the successful AESA algorithm [40] with a bounded-error estimation of the distances. We show that, the more available memory, the better the performance of the search process. Note that classical pivot-based algorithms do not have this feature.

We have shown experimentally that  $t$ -spanners are competitive against existing solutions. In particular we have shown that  $t$ -spanners are especially competitive in applications of interest to Information Retrieval: strings under edit distance and documents under cosine distance. For example, in an approximate string matching scenario typical of text databases, we show that  $t$ -spanners provide better space–time trade-offs than classical pivot-based solutions. Moreover,  $t$ -Spanners permit approximating AESA, which is an unbeaten index, within 1.5 times its distance evaluations using only about 3% of the space AESA requires. This becomes a feasible approximation to AESA, which in its original form cannot be implemented in practice because of its quadratic memory requirements. Furthermore, for document retrieval in a text database, we just perform 1.09 times the distance evaluations of AESA, using only 4% of its memory requirement.

To complete this approach, practical  $t$ -spanner construction algorithms are required for  $1.0 < t \leq 2.0$ . To the best of our knowledge, no previous technique had been shown to work well under this scenario (complete graph, metric distances, small  $t$ , practical construction time) and no practical study had been carried out on the subject. Our algorithms not only close this gap, but they are also well suited to general graphs.

We have shown that it is possible to build good-quality  $t$ -spanners in reasonable time. We have empirically obtained time costs of the form  $C_c \cdot n^{2+\frac{0.1\dots 0.2}{t-1}}$  and number of edges of the form  $C_e \cdot n^{1+\frac{0.1\dots 0.2}{t-1}}$ . Note that just scanning all the edges of the complete graph requires  $O(n^2)$  time. Moreover, just computing all the distances in a general graph requires  $O(n^3)$  time. Compared to existing algorithms, our contribution represents in practice a large improvement over the current state of the art. Note that in our case we do not provide any guarantee in the number of edges. Rather, we show that in practice we gen-

erate  $t$ -spanners with few edges with fast algorithms. Among the algorithms proposed, we have selected the recursive one as the most appropriate to index a metric database.

One of the most important advantages of this methodology is that the  $t$ -spanner naturally adapts, by its construction methodology, to the spatial distribution of the data set, which allows obtaining better performance both in the construction and the search phase (in particular, better than pivot-based algorithms). We have given enough empirical evidence to support the conjecture that this good behavior holds in any clustered space. Most real-world metric spaces have clusters, so we expect that  $t$ -spanners will also perform well in other real applications.

Finally, we have addressed the problem of indexing a dynamic database using  $t$ -spanners. We proposed mechanisms for object insertion and deletion, and  $t$ -spanner remodeling, that make up a robust method for maintaining the  $t$ -spanner up to date while preserving its quality. For insertions, the use of the incremental algorithm is efficient and yields  $t$ -spanners of good quality. For deletions, we obtain good results by combining lazy deletion (where the object itself is indeed removed) and periodic remodeling of the structure.

Several lines of future work remain open, both in  $t$ -spanner construction and in its use as a search tool:

- A first one is that, for search purposes, we do not really want the same stretch  $t$  for all the edges. Shorter edges are more important than longer ones, as Dijkstra's algorithm tends to use shorter edges to build the shortest paths. Using a  $t$  value that depends on the distance to estimate may give us better space-time trade-offs.
- We can consider fully dynamic  $t$ -spanners, which means that the  $t$ -spanner allows object insertions and deletions while preserving its quality without need of periodical remodeling. This is important in real-time applications where there is no time for remodeling.
- A weakness of our current construction algorithms is the need to have an externally computed model predicting their final number of edges. We are working on versions that use the  $t$ -spanner under construction to extrapolate its final size. Preliminary experiments show that the results are as good as with external estimation.
- Another line of work is probabilistic  $t$ -spanners, where most distances are  $t$ -estimated, so that with much fewer edges we find most of the results.
- Yet another idea is that we can build a  $t$ -spanner and use it as a  $t'$ -spanner, for  $t' < t$ . This may lose some relevant elements but improves the search time. The result is a probabilistic algorithm, which is a new successful trend in metric space searching [15,11,13]. In particular, we have observed that, in order to build a  $t$ -spanner, many distances are estimated better than  $t$

times the real one, so this idea seems promising. For example, a preliminary experiment in the string metric space shows that, with a 2.0-spanner and using  $t' = 1.9$ , we need only 53% of the original distance computations to retrieve 92% of the result.

- Finally, another idea is to use the  $t$ -spanner as a navigational device. A pivot is much more effective if it is closer to the query, as the ball of candidate elements has much smaller volume. We can use the  $t$ -spanner edges to start at a random node and approach the query by neighbors.

## 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.
- [3] R. Baeza-Yates. Searching: An algorithmic tour. In A. Kent and J. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 331–359. Marcel Dekker, New York, 1997.
- [4] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. 5th Combinatorial Pattern Matching (CPM'94)*, number 807 in LNCS, pages 198–212, 1994.
- [5] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.
- [6] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *Proc. 30th Symp. on the Theory of Computing (STOC'98)*, pages 161–168, 1998.
- [7] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.
- [8] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 357–368, 1997.
- [9] S. Brin. Near neighbor search in large metric spaces. In *Proc. 21st Conference Very Large Databases (VLDB'95)*, pages 574–584. Morgan Kaufmann, 1995.
- [10] W. Burkhard and R. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, 1973.
- [11] B. Bustos and G. Navarro. Probabilistic proximity searching algorithms based on compact partitions. *Journal of Discrete Algorithms*, 2(1):115–134, 2004.

- [12] 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.
- [13] E. Chávez, K. Figueroa, and G. Navarro. Proximity searching in high dimensional spaces with a proximity preserving order. In *Proc. 4th Mexican Intl. Conf. on Artificial Intelligence (MICAI'05)*, LNAI 3789, pages 405–414, 2005.
- [14] E. Chávez, J. Marroquín, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications*, 14(2):113–135, 2001.
- [15] E. Chávez and G. Navarro. Probabilistic proximity search: Fighting the curse of dimensionality in metric spaces. *Information Processing Letters*, 85:39–46, 2003.
- [16] E. Chávez, G. Navarro, R. Baeza-Yates, and J.L. Marroquín. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.
- [17] T. Chiueh. Content-based image indexing. In *Proc. 20th Conf. on Very Large Databases (VLDB'94)*, pages 582–593, 1994.
- [18] E. Cohen. Fast algorithms for constructing  $t$ -spanners and paths with stretch  $t$ . *SIAM Journal on Computing*, 28:210–236, 1998.
- [19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.
- [20] E. W. Dijkstra. A note on two problems in connexion with graphs. In *Numerische Mathematik*, volume 1, pages 269–271. Mathematisch Centrum, Amsterdam, The Netherlands, 1959.
- [21] D. Eppstein. Spanning trees and spanners. In *Handbook of Computational Geometry*, pages 425–461. Elsevier, 1999.
- [22] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.
- [23] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
- [24] M. Goldwasser, J. Bentley, K. Clarkson, D. S. Johnson, C. C. McGeoch, and R. Sedgewick. The sixth dimacs implementation challenge: Near neighbor searches, January 1999.
- [25] J. Gudmundsson, C. Levkopoulos, and G. Narasimhan. Fast greedy algorithms for constructing sparse geometric spanners. *SIAM Journal of Computing*, 31(5):1479–1500, 2002.
- [26] J. M. Keil. Approximating the complete Euclidean graph. In *Proc. 1st Scandinavian Workshop in Algorithm Theory (SWAT'88)*, LNCS 318, pages 208–213, 1988.

- [27] G. Kortsarz and D. Peleg. Generating sparse 2-spanners. *Journal of Algorithms*, 17(2):222–236, 1994.
- [28] 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.
- [29] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (AESAs) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994.
- [30] G. Navarro and R. Paredes. Practical construction of metric  $t$ -spanners. In *Proc. 5th Workshop on Algorithm Engineering and Experiments (ALENEX'03)*, pages 69–81, 2003.
- [31] G. Navarro, R. Paredes, and E. Chávez.  $t$ -Spanners as a data structure for metric space searching. In *Proc. 9th Intl. Symp. on String Processing and Information Retrieval (SPIRE'02)*, LNCS 2476, pages 298–309, 2002.
- [32] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.
- [33] D. Peleg and A. Schaffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.
- [34] D. Peleg and J. Ullman. An optimal synchronizer for the hypercube. *SIAM Journal on Computing*, 18:740–747, 1989.
- [35] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.
- [36] J. Ruppert and R. Seidel. Approximating the  $d$ -dimensional complete Euclidean graph. In *3rd Canadian Conf. on Computational Geometry*, pages 207–210, 1991.
- [37] D. Shasha and T. Wang. New techniques for best-match retrieval. *ACM Transactions on Information Systems*, 8(2):140–158, 1990.
- [38] M. Thorup and U. Zwick. Approximate distance oracles. In *Proc. 33rd Symp. on Theory of Computing (STOC'01)*, pages 183–192, 2001.
- [39] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, 1991.
- [40] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.
- [41] M. A. Weiss. *Data Structures and Algorithm Analysis*. Addison-Wesley, 2nd edition, 1995.
- [42] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms (SODA'93)*, pages 311–321. SIAM Press, 1993.

- [43] P. Yianilos. Excluded middle vantage point forests for nearest neighbor search. Technical report, NEC Research Institute, 1998. In *6th DIMACS Implementation Challenge: Near Neighbor Searches Workshop, ALENEX'99*.