

# Spanner Approximations in Practice

Markus Chimani  

Theoretical Computer Science, Universität Osnabrück, Germany

Finn Stutzenstein 

Theoretical Computer Science, Universität Osnabrück, Germany

---

## Abstract

A multiplicative  $\alpha$ -spanner  $H$  is a subgraph of  $G = (V, E)$  with the same vertices and fewer edges that preserves distances up to the factor  $\alpha$ , i.e.,  $d_H(u, v) \leq \alpha \cdot d_G(u, v)$  for all vertices  $u, v$ . While many algorithms have been developed to find good spanners in terms of approximation guarantees, no experimental studies comparing different approaches exist. We implemented a rich selection of those algorithms and evaluate them on a variety of instances regarding, e.g., their running time, sparseness, lightness, and effective stretch.

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Sparsification and spanners; General and reference  $\rightarrow$  Experimentation

**Keywords and phrases** Graph spanners, experimental study, algorithm engineering

**Digital Object Identifier** 10.4230/LIPIcs.ESA.2022.37

**Related Version** *Extended Version*: <https://arxiv.org/abs/2107.02018>

## 1 Introduction

Consider a directed or undirected graph  $G = (V, E)$  with  $n$  vertices and  $m$  edges. The distance  $d_G(u, v)$  is the length of a shortest path between two vertices  $u$  and  $v$ , possibly subject to edge weights  $w: E \rightarrow \mathbb{R}^+$ . A *spanner* is a sparse subgraph of  $G$  that preserves these distances to some quality degree. Spanners were introduced by Peleg and Schäffer [38] after the first mention by Awerbuch [5]. There are many spanner variants, see Ahmed et al [1] for a survey. In this paper we are going to focus on the probably most popular variant:

► **Definition 1** (Multiplicative  $\alpha$ -Spanner [38]). *Given a directed or undirected graph  $G = (V, E)$  and a stretch  $\alpha \geq 1$ , a multiplicative  $\alpha$ -spanner  $H = (V, E')$  is a subgraph of  $G$  with  $E' \subseteq E$  such that  $d_H(u, v) \leq \alpha \cdot d_G(u, v)$  holds for all pairs  $(u, v) \in V \times V$ .*

For simplicity we use the term *spanner* for the multiplicative  $\alpha$ -spanner in the following.

Given a graph  $G$ , finding any spanner is trivial since  $G$  is a spanner of itself. The problem we are interested in is to find a *good* spanner. Next to the stretch  $\alpha$  itself, there are three basic measures of a spanner's quality: the *size* is the number of edges  $|E'|$ ; the *sparseness*  $s(H) = \frac{|E'|}{|E|}$  is the relative size w.r.t. the original graph, with a lower value indicating a sparser spanner; the *lightness*  $\ell(H) = \frac{W(E')}{W(\text{MST}(G))}$  with  $W(F) = \sum_{e \in F} w(e)$  compares the weight of the spanner to the weight a minimum spanning tree in  $G$ .

► **Definition 2** (Sparsest (Lightest)  $\alpha$ -Spanner). *Given a graph  $G = (V, E)$  and a stretch  $\alpha \geq 1$ , find a sparsest (lightest) multiplicative  $\alpha$ -spanner  $H = (V, E')$  such that  $s(H) \leq s(H')$  ( $\ell(H) \leq \ell(H')$ , respectively) for all other spanners  $H'$  of  $G$ .*

Finding good spanners is motivated by several applications, e.g., ranging from routing problems [5, 40, 17, 43] to distributed computing [39, 18]. Also many theoretical applications require spanners, e.g., approximate distance oracles [37, 45], almost shortest paths [25, 22], or access control hierarchies [31, 9]. Thus, while finding a sparsest or lightest spanner is NP-hard [38, 11], a multitude of algorithms was developed to find sparse or light spanners.



© Markus Chimani and Finn Stutzenstein;  
licensed under Creative Commons License CC-BY 4.0  
30th Annual European Symposium on Algorithms (ESA 2022).

Editors: Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman; Article No. 37; pp. 37:1–37:15  
Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We are interested in the computation of spanners in practice. Farshi and Gudmundsson [27] compared specialized algorithms based on WSPD-graphs [12] and  $\Theta$ -graphs [16, 32] for the geometric case; Bouts et al. [10] presented experiments, e.g., on multilevel subset additive spanners. Regarding exact algorithms, Sigurd and Zachariassen [44] developed in 2004 an exact spanner algorithm based on an ILP with an exponential number of variables, each representing a shortest path. Graphs with up to 64 vertices were tested but not all instances could be solved within the time limit of 30 minutes. Ahmed et al. [2] published a compact ILP based on multicommodity flow in 2019, which can be solved by standard B&B-techniques. They tested graphs with up to 100 vertices, but the solver needed up to 40 hours to solve the largest instances on a high-performance 400-node cluster. They did not compare the results to [44] or other popular heuristics. Apart from these studies, there seem to be no further experimental studies, in particular none for the arguably most prominent base case of multiplicative  $\alpha$ -spanners in non-geometric settings.

**Contribution.** Despite the multitude of applications, there is an obvious total lack of practical experience with polynomial algorithms for non-geometric spanners. While there are various intrinsically different algorithmic ideas known in literature, there is no understanding which of these concepts lend themselves to fruitful practical applications. The main focus of this paper is thus to gain some insight in the practical consequences of these algorithmically diverse ideas. – In Sect. 2, we provide an overview of the known algorithms. We argue our selection of algorithms and give implementation details in Sect. 3. In Sect. 4, we present the results of our experimental study.

## 2 Known Algorithms

Much work and effort has been put into developing different approaches to find spanners. We will categorize the existing literature into three main categories: greedy algorithms, approximation algorithms, and other methods. For a rich theoretical survey of graph spanners, their theoretical background and algorithms, see Ahmed et al. [1].

**Greedy algorithms.** One of the earliest algorithms is by Althöfer et al. [4] and is reminiscent of Kruskal’s algorithm. The algorithm ADDJS, often named *Basic Greedy Spanner Algorithm*, starts with a spanner without edges and adds edges  $\{u, v\}$  with increasing weight to the spanner, if the shortest path between  $u$  and  $v$  is currently too long, i.e.,  $d_H(u, v) > \alpha \cdot w(\{u, v\})$ . For a stretch  $\alpha = 2k - 1$  ( $k \in \mathbb{N}_{\geq 1}$ ), this algorithm creates a spanner of size  $\mathcal{O}(n^{1+1/k})$  and lightness  $\mathcal{O}(n/k)$ . The running time is mostly dominated by a Dijkstra run for each edge:  $\mathcal{O}(m(n^{1+1/k} + n \log n))$ . – Roditty and Zwick [42] sped up to the distance calculation in the spanner by incrementally maintaining a shortest-path tree, but lose any lightness guarantee in the process. Thorup and Zwick [45] explored *approximate distance oracles*, yielding a spanner algorithm as a byproduct, but as before, no lightness guarantee can be given.

To further improve the quality measures, more refined analyses using  $\varepsilon$ -parameterized algorithms were introduced. Thereby, one allows the spanner to violate the stretch requirement by a factor of up to  $1 + \varepsilon$ , while the approximation ratio of the sparsity and lightness is measured against the best spanner not violating this requirement. Chandra et al. [13], Elkin and Solomon [26] and Elkin et al. [24] use various techniques (e.g., dynamic shortest-path trees, further auxiliary graphs, and distance oracles) and refined analyses to create a spanner with guarantees for size and lightness as well as maintaining a low running time. Alstrup et al. [3] provide the currently best guarantees; their algorithm creates a  $((2k - 1)(1 + \varepsilon))$ -spanner

with size  $\mathcal{O}_\varepsilon(n^{1+1/k})$  and lightness  $\mathcal{O}_\varepsilon(n^{1/k})$  within a running time of  $\mathcal{O}(n^{2+1/k+\varepsilon'})$ . Thereby,  $\mathcal{O}_\varepsilon$  ignores polynomial factors depending on  $1/\varepsilon$ . However, ADDJS still has the best lightness guarantee while strictly providing an  $\alpha = 2k - 1$  stretch and is existentially optimal, see Ahmed et al. [1, Chapter 4] for a more thorough theoretical insight. Additionally, most above modifications lead to very complex algorithms that are often not practically implementable due to the intricate additionally required data structures mentioned above.

**Clustering algorithms.** Baswana and Sen [6] (BS) gave a method based on growing clusters around sampled vertices. It is special compared to all other algorithms as it does not do any local or global distance calculation. It computes an  $\alpha = 2k - 1$  spanner for a weighted, undirected graph in two phases. First, there are  $k - 1$  rounds of growing clusters by sampling vertices and successively adding nearest neighbors to each cluster. Then, clusters are joined by adding all non-clustered vertices to the nearest adjacent cluster and interconnecting all clusters. This yields a spanner of size  $\mathcal{O}(kn^{1+1/k})$  in an expected running time of  $\mathcal{O}(km)$ , but no lightness guarantee. For an unweighted graph, the size is bounded by  $\mathcal{O}(n^{1+1/k} + kn)$ .

**Approximation algorithms.** An early  $\mathcal{O}(\log(m/n))$ -approximation in terms of size by Kortsarz and Peleg [34] was published in 1994 for the special case of a 2-spanner for unweighted and undirected graphs (algorithm KP). An edge  $e$  is *covered*, if it is not part of the spanner, but part of a triangle in the original graph and both other edges are in the spanner, so the spanner property holds for  $e$ . The idea is to cover a large number of edges while not adding too many edges to the spanner by finding dense subsets of vertices and adding connecting edges to the spanner. KP calculates *dense subsets*  $U_v$  of neighboring vertices  $N(v)$  for each vertex  $v$  and continues with the densest subset, say  $U_w$ . The star  $\{\{u, w\} | u \in U_w\}$  is added to the spanner. The edges within the dense subset are now covered and finding the densest subgraph allows for covering the most edges. This is repeated while there exists a dense subset with a density larger than 1. The running time is bounded by  $\mathcal{O}(nm \cdot \text{MDS}(n, m))$  where  $\text{MDS}(n, m)$  is the running time of the algorithm to solve the maximum density subgraph problem. Using Goldberg's algorithm [30], the resulting time complexity is  $\mathcal{O}(m^2 n^3 \log n)$ . There exist more advanced algorithms for the maximum density problems so the theoretical running time can be lowered to  $\mathcal{O}(m^2 n^2 \log(n^2/m))$  [28].

For undirected graphs we know of no explicit further approximation results. One reason may be that ADDJS already has an  $\mathcal{O}(n^{1+1/k})$  guarantee for the spanner size. For a connected original graph, a spanner must have at least  $n - 1$  edges, as it has to be connected as well. This lower bound for an optimal solution thus results in a straight-forward  $\mathcal{O}(n^{1/k})$ -approximation in terms of size for every undirected graph.

For directed graphs, Elkin and Peleg [25] gave the first  $\tilde{\mathcal{O}}(n^{2/3})$ -approximation in the special case  $\alpha = 3$ . For general  $\alpha \geq 3$ , Bhattacharyya et al. [9] published an  $\tilde{\mathcal{O}}(n^{1-1/\alpha})$ -approximation combining two techniques: The spanner is the union of the arc sets of two graphs. The first graph is obtained by solving an LP relaxation and rounding the solution. The second one is created by sampling vertices and growing BFS arborescences from them. Dinitz and Krauthgamer [20] improved the techniques and Berman et al. [8] gave the currently best solution for general stretch  $\alpha \geq 1$  (algorithm BBMR): They introduced an  $\mathcal{O}(n^{1/2} \log n)$ -approximation for weighted, directed graphs. For the special case  $\alpha = 3$  (later extended to  $\alpha = 4$  by Dinitz and Zhang [21]) a slightly modified approach can be taken to achieve an  $\tilde{\mathcal{O}}(n^{1/3})$ -approximation for directed and unweighted graphs. The techniques used are similar to the approach of Bhattacharyya et al. [9]: The arcs of the original graphs are categorized as either *thin* or *thick* by the number of shortest paths connecting the endpoints of the

■ **Table 1** Considered algorithms. The letters G, A, C and P in the second column stand for greedy, approximation, clustering, and probabilistic, respectively. \*Assume  $\alpha = 2k - 1$ ,  $k \in \mathbb{N}_{\geq 1}$ .

Algorithm		$w(e)$	Stretch	Spanner Properties*	Running time
ADDJS [4]	G	✓	$\alpha \in \mathbb{R}_{\geq 1}$	$s(H) \in \mathcal{O}(n^{1+1/k})$ $\ell(H) \in \mathcal{O}(n/k)$ [see text]	$\mathcal{O}(m(n^{1+1/k} + n \log n))$
KP [34]	A	×	$\alpha = 2$	$\mathcal{O}(\log(m/n))$ -approx. in size	$\mathcal{O}(m^2 n^3 \log n)$
BBMRY [8]	A	✓	$\alpha \in \mathbb{R}_{> 1}$	$\mathcal{O}(n^{1/2} \log n)$ -approx. in size	poly
BS [6]	C	×	$\alpha \in \mathbb{N}_{\geq 3}$ , odd	$s(H) \in \mathcal{O}(n^{1+1/k} + kn)$	expected $\mathcal{O}(km)$
BS [6]	C	✓	$\alpha \in \mathbb{N}_{\geq 3}$ , odd	$s(H) \in \mathcal{O}(kn^{1+1/k})$	expected $\mathcal{O}(km)$
EN [23]	P	×	$\alpha \in \mathbb{N}_{\geq 1}$ , odd	$s(H) \in \mathcal{O}(n^{1+1/k}/\varepsilon)$ $0 < \varepsilon < 1$	expected $\mathcal{O}(m)$ , success prob. $\geq 1 - \varepsilon$

arc. An LP rounding covers all thin arcs, and growing arborescences from sampled vertices cover all thick arcs. For the (I)LP, *antispanners* were introduced. Given an arc  $(u, v)$ , an antispanner is an arc set  $A$ , such that no shortest path with a distance less than  $\alpha \cdot d_G(u, v)$  exists between  $u$  and  $v$  in  $G \setminus A$ .  $A$  is a minimal antispanner for  $(u, v)$  if no  $A' \subset A$  is an antispanner for  $(u, v)$ . The ILP tries to cover arcs so that at least one arc of each minimal antispanner is used. If this is achieved, the selected arcs resemble a feasible solution. Due to the exponential number of minimal antispanners, we require a separation oracle to solve the LP relaxation of this ILP with cutting planes. The separation oracle has an exponentially small probability in  $n$  of failing.

**Randomized Algorithms.** A probabilistic method was presented by Elkin and Neiman [23] to calculate a spanner with stretch  $2k - 1$  for undirected, unweighted graphs (algorithm EN). They built upon Miller et al. [36, 35] and improved the size to  $\mathcal{O}(n^{1+1/k}/\varepsilon)$ . For each vertex  $u$ , a random value  $r_u$  is drawn from an exponential distribution. Each vertex broadcasts a message with  $r_u$  as the content to all vertices within distance  $k$ . A vertex  $x$ , receiving messages from some vertices  $U \subset V$ , stores  $m_u(x) = r_u - d_G(u, x)$  for each vertex  $u \in U$  together with an edge adjacent to  $x$  that lies on a shortest path between  $u$  and  $x$ . The edges belonging to the messages with  $m_u(x) \geq \max_{u \in V} \{m_u(x)\} - 1$  are added to the spanner. Since the algorithm has a success probability of at least  $1 - \varepsilon$ , we can obtain an expected running time of  $\mathcal{O}(m)$  by iterating.

### 3 Considered Algorithms and Implementation Details

As motivated in Sect. 1, we aim at covering very diverse algorithmic approaches and investigating their relative merits, rather than several algorithmic variants finetuning a common idea. Only once an understanding on the most worthwhile algorithmic concepts has been achieved, deeper comparisons of algorithmic variants would seem worthwhile.

Thus, we select five, intrinsically different, algorithmic approaches to analyze; Table 1 provides an overview of the characteristics of each algorithm. With the goal of small running times and larger instances, we decided against reestablishing the performance of the exact algorithms [44, 2]; as summarized in Sect. 1, their running times are nowhere competitive on the instances we aim to consider. As the representative for the fundamental greedy algorithms, we selected ADDJS. While the algorithms of Thorup and Zwick [45] and Roditty and Zwick [42] improve the running time, ADDJS promises better results w.r.t. size and lightness. As such, there was no good argument to consider these or the algorithmically

and implementationwise much more involved  $\varepsilon$ -parameterized variants at this stage of the scientific knowledge discovery (but see Sect. 5). This argument is further amplified by the fact that many of these variants require very sophisticated subalgorithms (see above) which are by themselves unknown territory from the algorithm engineering viewpoint.

From the approximation algorithms we selected KP and BBMRV as they are the state-of-the-art. For BBMRV, we implemented the variant for arbitrary  $\alpha$ , not the special improvements for  $\alpha = 3$  and  $\alpha = 4$ . In the following, we list some implementation specific details; all our implementations guarantee the running time of the algorithmic descriptions.

**Althöfer et al. (ADDJS).** We include a straightforward observation to speed-up the computation: We bound the shortest-path search for each pair of vertices by  $\alpha \cdot w(\{u, v\})$  since finding a longer path (or no path at all) results in adding the edge  $\{u, v\}$  to the spanner. Note that one is not limited to a stretch  $\alpha \in \mathbb{N}_{\text{odd}}$ ; the algorithm works for arbitrary  $\alpha \geq 1 \in \mathbb{R}$ , but may only provide the size and lightness guarantees for the next odd integer.

**Baswana and Sen (BS).** We broadly follow the detailed algorithmic description of [6]. We changed only one small aspect: The second and third step of the first phase can be combined in a single pass. After sampling clusters, it is not necessary to first calculate the nearest neighboring sampled cluster for each vertex and store it in some data structure because the third step can directly follow and does not depend on other vertices.

**Kortsarz and Peleg (KP).** The algorithm mainly depends on an implementation for the *maximum density subgraph problem*. We use Goldberg's algorithm [30] due to its simplicity of implementation. It resembles a binary search with  $\mathcal{O}(\log n)$  steps: A source and target vertex  $(s, t)$  are added to the graph and connected to all original vertices. In each step the weights of the new edges are chosen depending on the current search value. A minimum  $s$ - $t$ -cut is calculated and based on the resulting cut it is decided, whether the lower or upper half of the search interval is used. To solve the minimum  $s$ - $t$ -cut problem we use Goldberg-Tarjan's max-flow algorithm [29] with global relabeling and gap relabeling heuristic requiring  $\mathcal{O}(n^2m)$  time. This results in a running time for Goldberg's maximum density subgraph algorithm of  $\mathcal{O}(n^2m \log n)$  and, as described above, in a complete running time of  $\mathcal{O}(m^2n^3 \log n)$  for KP.

The algorithm has to maintain three sets  $H^s$ ,  $H^c$ , and  $H^u$  for the spanner edges, covered edges, and uncovered edges. In the implementation it is sufficient to only use the original graph's edge set  $E = H^u$  (by shrinking  $G$  on the fly) and the spanner's edge set  $E' = H^s$ . There is no need to explicitly store covered edges  $H^c$  in some data structure. For each maximum dense subset  $U_w$ , the star edges are added to  $E'$ . The star edges and the edges of the implied dense subset graph  $E(U_w) = \{\{u, v\} \in E \mid u, v \in U_w\}$ , which are the covered edges, are removed from  $E$ , so that the covered edges are simply entirely removed. Since we have to calculate the density of the maximum density subset, we have  $E(U_w)$  already available, so no extra work is required to calculate the edges induced by the subset of vertices.

**Berman et al. (BBMRV).** The original algorithmic description leaves some implementation details open. First, the property of an arc  $(s, t)$  being *thin* or *thick* has to be efficiently evaluated [8, Definition 2.2]. The definition uses a *local graph*  $G^{s,t} = (V^{s,t}, E^{s,t})$  induced by all vertices belonging to paths from  $s$  to  $t$  with a length of at most  $\alpha \cdot d_G(s, t)$ . An arc  $(s, t)$  is thick, if  $|V^{s,t}| \geq n^{1/2}$ . Calculating all possible paths is not an option, so in- and out-arborescences can be used. First, we precalculate these arborescences since we need them for evaluating the local graphs and during sampling afterwards. For each vertex  $r$ , an in-

and out-arborescence rooted at  $r$  is calculated. The distances to other vertices  $x$  are saved in  $d_{\text{in}}(r, x)$  ( $d_{\text{out}}(r, x)$ , respectively) together with the respective predecessor to be able to access the actual trees during sampling. To check if  $|V^{s,t}| \geq n^{1/2}$  holds, we can use the equivalence:  $x \in V^{s,t} \iff d_{\text{out}}(s, x) + d_{\text{in}}(t, x) \leq \alpha \cdot d_G(s, t)$ . Checking each vertex  $x$ , we count  $|V^{s,t}|$  and decide the property for each arc. This could be done by using either an in- or out-arborescence, but both types are needed for the sampling, so we can precalculate both anyhow. Consequently, the subsequent sampling does not involve any shortest-path calculations and can be done by looking up predecessor relations.

Next, given an arc  $(s, t)$  it must be checked, if a set  $R \subseteq E$  settles the arc [8, Definition 2.3], i.e.,  $R$  satisfies the  $\alpha$ -spanner property for  $(s, t)$ . This is done by running a bounded Dijkstra on  $R$  with a running time of  $\mathcal{O}(|R| + n \log n)$ . The check whether an arc is settled is done quite often, e.g., during the separation method and in the minimization of an antispanner.

Finally, let us focus on the separation method and the creation of antispanners. The former is straightforward: After rounding the fractional solution, obtaining  $R$ , we check every thin arc, whether it is settled by  $R$ . For an unsettled thin arc  $(s, t)$ , we create an antispanner w.r.t.  $R$  (see [8, Claim 2.4]). Similar to the thin arc property where the vertices  $V^{s,t}$  are required, we need the arc set  $E^{s,t}$ , as  $A = E^{s,t} \setminus R$  is an antispanner. An arc  $(u, v)$  is an element of  $E^{s,t}$  if and only if  $d_{\text{out}}(s, u) + w(u, v) + d_{\text{in}}(t, v) \leq \alpha \cdot d_G(s, t)$ ; this check can reuse the precalculated arborescences. After finding the antispanner it has to be minimized. We greedily remove arcs from  $A$ , while the unsettled thin arc remains unsettled. If no further arc can be removed,  $A$  is a minimal antispanner.

**Elkin and Neiman (EN).** Provided the very compact description of the algorithm and no explicit pseudocode, one has to be very careful not to gloss over intricate important details. We implemented the algorithm in the standard centralized model. The authors also highlight the distributed and PRAM models (as the algorithm can be parallelized well), but we aim to keep it comparable to all other algorithms.

To start, remember that the edges saved together with the received messages must belong to *some* shortest  $u$ - $x$ -path. Naturally, we store the edge where  $x$  received the message from. To ensure that the messages travel along shortest paths, we use a breadth-first-search (BFS) starting at  $u$  with a depth limit of  $k$ , without the need of a Dijkstra computation.

The algorithm includes two feasibility checks. If one fails, the algorithm does not provide a feasible solution. The first check, that  $r_u < k$  holds for all  $u \in V$ , can be done directly while generating values  $r_u$  to let the algorithm fail quickly if the property does not hold. The second check at the end of the algorithm asserts that there are sufficiently many edges. Elkin and Neiman state that the spanner must have at least  $n - 1$  edges and the algorithm fails, if  $|E'| < n - 1$ . This is only correct if the original graph is connected. If it has  $c \geq 2$  components, we can use the condition  $|E'| < n - c$  instead.

A major improvement in terms of memory consumption can be made when reversing the way the algorithm is formulated. The original formulation can chiefly be described as:

1. For each  $u \in V$ , broadcast messages  $m_u(x)$  to every vertex  $x$  within distance  $k$ .
  2. For each receiving  $x \in V$ : Calculate  $\max_{u \in V} \{m_u(x)\}$  and add edges to the spanner.
- Following this order, one has to save two  $n \times n$  matrices containing the message values  $m_u(x)$  and the edge where the messages are received. In a pilot study, this resulted in out-of-memory errors for very sparse graphs ( $m \approx 2n$ ) with  $n \approx 40,000$ , which other algorithms can handle. To lower the memory consumption, we reversed the logic to focus on each receiving vertex  $x$ : A BFS starting at  $x$  identifies all vertices  $u$  that can broadcast a message to  $x$ . Then, the aggregation and edge addition can directly be done afterwards, before proceeding with the next  $x$ . To summarize all improvements, the code is provided in [15, Appendix A].

We note that it is possible to provide values for  $\varepsilon$  that are not in the interval  $(0, 1)$ . Especially  $\varepsilon \geq 1$  can produce good spanners in practice with the downside of an increasing possibility to fail. For results of a pilot study to choose  $\varepsilon$ , see [15, Appendix B].

## 4 Experimental Results

We are now ready to present the main content of this paper. The considered algorithms are tested with a variety of instances and parameter settings, as described in the following. In Sect. 4.1 we take a look at the running time of each algorithm. We compare the quality of the resulting spanners in Sect. 4.2. Finally, in Sect. 4.3, we propose and briefly investigate two further graph measures we deem interesting in their own right: We ask about the mean degrees and the mean lengths of the shortest paths in weighted spanners (compared to the input graphs). Investigating these measures can help us to better understand the different behaviors and results of the various algorithms.

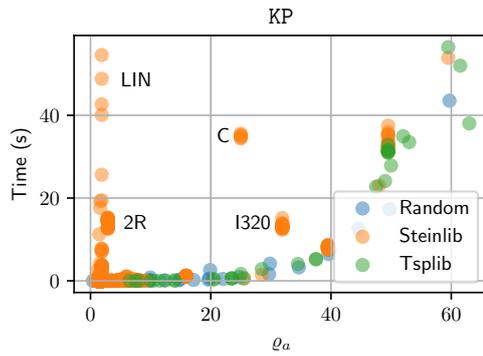
**Setup.** Since BBMR is the only algorithm that would work (with known quality guarantees) on directed graphs, we consider only undirected graphs. Edge weights are often a requirement in applications, but not all algorithms can take weighted graphs as inputs; thus we use each graph twice, once with and once without weights. In the following we will use the absolute density  $\varrho_a(G) = m/n$  and the relative density  $\varrho_r(G) = m/\binom{n}{2}$  to categorize graphs. In the experiments we consider integer stretches 2, 3, 4, 5, and 7. To test a variety of graph types, three instance libraries are used:

1. *Steinlib* [33]: This well-known library was originally created for the Steiner tree problem, but has also been used in many studies for related problems. Since several application areas of Steiner trees and spanners overlap, it seems judicious to consider this set. It contains 1207 graphs in 44 subsets. Over 62% of the graphs have  $\varrho_r = m/\binom{n}{2} < 2.5\%$  (mostly sparse). On average, the graphs have 1189 vertices and 8549 edges; the largest graph has 38418 vertices.
2. *Tsplib* [41]: The well-established Tsplib contains complete weighted graphs. We omitted 27 of the 122 graphs as they have edge weights 0 or are directed. About half of the graphs have  $< 200$  vertices; the largest graph has 7397 vertices. The graphs have 771 (250) vertices on average (median).
3. *Random*: We generated random Erdős-Rényi graphs with  $|V| \in \{10, 20, 50, 100, 200, 500, 1000, 2000\}$ , a relative density of  $i/10$  for  $i \in \{1, 2, \dots, 9\}$  and 10 graphs of each combination. Apart from the unweighted instances, we also consider two different random weight function: weights in the range  $1 \pm 1/3$  (**W1**), and integer weights between 1 and  $n$  (**W2**). The former ensures that each edge is a shortest path between its end vertices, as is the case for unweighted instances, whereas the latter includes “obviously superfluous” edges. All graphs are available at [7].

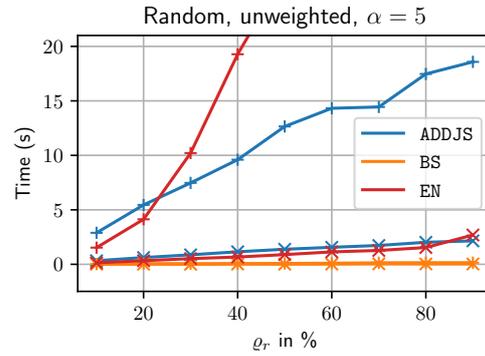
All implementations are freely available and will be part of the next release of the open source C++ *Open Graph algorithms and Datastructures Framework* [14] ([www.ogdf.net](http://www.ogdf.net)). All experiments were performed on an Intel Xeon Gold 6134 with 256 GB RAM under Debian 10.2 using gcc 8.3.0-6 (-O3). For BBMR, we use CPLEX 20.1 [19] as the LP solver. When conducting an experimental study, and i.p. due to the page restriction, one has to decide between in-depth analyses of a smaller set of certain aspects or a broader approach of reporting the overall algorithmic behaviors over a larger field of questions and parameterizations. As our goal herein is to broadly assess competing algorithmic ideas, the second option makes more sense. However, all detailed data of our experiments is available at [7] for other researchers (or a subsequent longer journal version) to investigate further.

■ **Table 2** Solved instances per algorithm and average time for successful cases.

Library	ADDJS	BS	KP	BBMR	EN
Random	98.62% (2.96 s)	100.00% (0.03 s)	66.67% (2.46 s)	70.16% (3.07 s)	100.00% (3.10 s)
Steinlib	99.88% (0.48 s)	100.00% (0.11 s)	91.55% (1.91 s)	83.79% (5.32 s)	100.00% (0.37 s)
Tsplib	81.75% (4.98 s)	98.71% (1.47 s)	33.01% (14.75 s)	60.74% (5.70 s)	76.05% (3.11 s)
All	98.46% (0.92 s)	99.91% (0.21 s)	86.02% (2.30 s)	80.91% (5.21 s)	98.20% (0.66 s)



■ **Figure 1** Instances solved by KP within the time limit for each instance library. There are no further solved instances for larger  $\rho_a$ .



■ **Figure 2** Comparing ADDJS, BS, and EN for  $n = 500$  ( $\times$ ) and  $n = 1000$  ( $+$ ). BBMR cannot solve any of these instances.

**Parameterizing the Randomized Algorithms.** To fairly evaluate the randomized algorithms BS and EN, we first need to investigate their parameterizations. We report on these additional experiments in [15, Appendix B]. In the following, we use the most competitive settings of 1000 iterations for BS and 200 iterations for EN with  $\varepsilon = 0.8$ .

#### 4.1 Running Time

We start with investigating the running times of our algorithms. We set a time limit of 60 seconds for each calculation: ADDJS, BS, and EN can solve the majority of all instances within this limit (cf. Table 2), and an additional pilot study shows that higher limits of 90 or 120 seconds do not significantly increase the number of solved instances. Before comparing the algorithms, we discuss their runtime behavior individually. In the following, when speaking about averages we consider the standard arithmetic mean, unless specified otherwise.

**ADDJS.** The running time is linear in  $m$  for fixed  $n$ . Almost all Steinlib instances can be solved within 6 seconds and all Tsplib instances with  $n \leq 1200$  within the time limit. The densest instances over all libraries with  $\rho_r \geq 90\%$  and  $n \approx 1000$  can barely be solved within the time limit. The running time shows a dependency on the graph weights: unweighted graphs can be solved 33.45% faster than weighted ones. Furthermore, higher stretches are faster to compute. There is one peculiar outlier: while all other combinations of weights and stretches allow all Random instances to be solved within the time limit,  $\alpha = 2$  for weighted (and **W1**-weighted, see also Sect. 4.2) instances is drastically slower and only achieves a success rate of 89%. This is also the only case where **W1** yields higher running times than **W2** and the unweighted case. Over the solved unweighted instances,  $\alpha = 2$  is 42.61% slower than  $\alpha = 3$  (while, e.g.,  $\alpha = 3$  is only 6.19% slower than  $\alpha = 4$ ).

**BS.** The clustering approach is very fast. Every instance from Random is solved in under 0.6 seconds. The running time behaves linearly in  $m$ , as expected. Analogous to ADDJS, unweighted instances are faster, but in contrast, a *lower* stretch is faster as well. For Random, the running times for **W1** are in-between those of **W2** and unit weights. Almost all instances of the Tsplib can be solved; only the two largest instances ( $n = 5915, 7397$ ) exceed the time limit for  $\alpha \geq 5$ . The majority of the Steinlib instances can be solved within a second. Generally, BS seems to have difficulties with large, very sparse graphs; for these graphs with  $\varrho_r \leq 0.01\%$ , the running time is up to 17.5 seconds (cf. [15, Appendix, Figure 8]). These are Steinlib instances from the subsets LIN, ALUT, and ALUE (grid graphs with holes,  $n \geq 20000$ ).

**KP.** Surprisingly, there is no clear relationship between the running time and  $n$ ,  $m$ , or  $\varrho_r$ ; however the absolute density is limiting: For instances with  $40 \lesssim \varrho_a \lesssim 60$ , the running time drastically starts to increase in comparison to  $\varrho_a \lesssim 40$  and no instances with  $\varrho_a \gtrsim 60$  can be solved in time (cf. Fig. 1). This behavior is consistent over all three libraries except for a handful of outliers from Tsplib. Regarding Steinlib, the sets 2R, I320, C, PUC and the largest instances of LIN require higher running times due to the very large  $n$ . The complete graphs of Tsplib can be solved with up to 120 vertices, which corresponds to  $\varrho_a = 59.5$ .

**BBMR.** Its running time depends on  $\alpha$  and the weights: Similar to ADDJS, unweighted graphs are slightly faster (by 10.28%), but a higher stretch results in a lower running time. It also shares the performance degradation for  $\alpha = 2$  on unweighted graphs. Overall, both the timeout rates and the running times are very high. Regarding the instance sizes, the number of edges has a disproportionate impact on the running time, i.e., only Steinlib instances with  $m \leq 5000$  edges can be solved, regardless of the number of vertices. This is not surprising since the antispanner creation and minimization depends on the original graph's edges. Graphs from Random and Tsplib can only be solved if they have less than 250 nodes. Even though BBMR has a probability of failing, we never observed any failure during the experiments.

**EN.** We observe a linear running time in  $m$ . We measured the time until the first success, so previous failures are included in the running times. EN can solve graphs with up to 1000 vertices for Random and Tsplib, with a smaller stretch being slightly faster, as the BFS depth depends on  $\alpha$ . The maximum time for Steinlib is 14 seconds.

**Comparison.** Table 3 shows the percentage of instances an algorithm can solve strictly faster than another algorithm, as well as the average relative speed-up factor in these cases. An example is provided in Fig. 2 for unweighted instances and  $\alpha = 5$ . Clearly, BBMR is the slowest of the considered algorithms. Only KP was sometimes slower (in 8.2% of the cases, roughly 14-fold); but when KP was faster, it was so by a factor of over 116. BS shows a significant number of instances where it is fastest, as it benefits from traversing adjacency lists instead of calculating distances. Also, its speed-up factor in these cases is rather strong. EN and ADDJS both share a similar number of instances each can solve faster. This is no surprise since EN's BFS for each node is similar to the Dijkstra runs for ADDJS in the unweighted setting. Only for large or dense instances, ADDJS gets noticeably faster than EN (cf. Fig. 2). If these values were to be restricted to larger or denser graphs (e.g.  $n \geq 100$  or  $\varrho_r \geq 5\%$ ), the results would be even clearer due to the fact that all small/sparse instances can be solved within fractions of seconds by most algorithms.

■ **Table 3** Percentage of instances where A was strictly faster than B. “–” denotes that the algorithms cannot be directly compared due to their incompatible stretch restrictions (2 vs. odd), see Table 1. The speed-up factor is given in parentheses. Only instances both algorithms can solve are included.

A \ B	ADDJS	BS	KP	BBMRY	EN
ADDJS		14.83% (1.45)	51.23% (123.61)	93.24% (210.88)	19.38% (1.07)
BS	36.07% (24.71)		–	92.99% (188.60)	28.70% (30.23)
KP	1.19% (1.27)	–		86.65% (116.61)	–
BBMRY	0.00% (–)	0.00% (–)	8.20% (14.17)		0.00% (–)
EN	27.10% (2.68)	11.52% (0.45)	–	93.37% (170.12)	

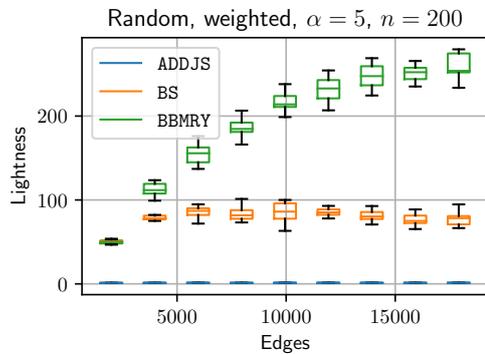
## 4.2 Quality

We may now consider the standard measures for spanners: their lightness and sparseness.

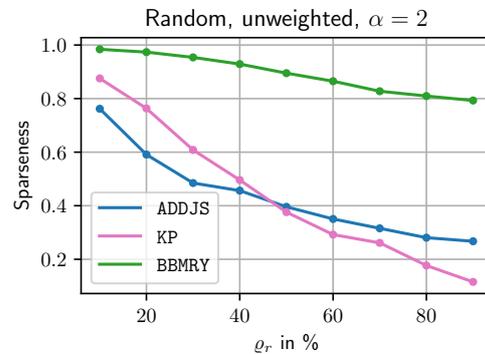
**Weight functions for the Random instances.** A key observation from the running times above also holds true for the solutions’ quality: The unweighted instances and the weighting **W1** behave very similarly. In any measure, **W1** lies between **W2** and the unweighted case, but is always very close to the latter; with increasing graph size, the sparseness of **W1** converges to that of the latter. Thus, in the following discussions regarding Random, we focus on unweighted instances and **W2**; all statements about the former are also true for **W1**.

**Lightness.** As a concept different from size, the lightness is only relevant for weighted instances. **BBMRY** and **BS** do not provide any guarantees, and indeed, both yield arbitrarily high lightness values in practice across all instance libraries (cf. Fig. 3). **BBMRY**’s lightness is on average 33.5 times higher than **ADDJS**’s; and **BS**’s lightness is 11.8 times higher. **BS** and **BBMRY** never yield a lower lightness than **ADDJS** on any instance. In summary, there is no alternative to **ADDJS** for applications requiring a lightness guarantee.

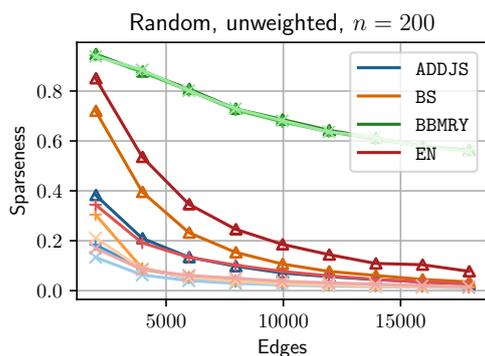
**Sparseness.** First, consider  $\alpha = 2$  and unweighted graphs. Surprisingly, **KP** yields a lower sparseness than **ADDJS** for  $q_r \gtrsim 50\%$  (cf. Fig. 4). **BBMRY** yields only high sparseness in comparison to those algorithms. On **Tsplib** (i.e., complete graphs), **KP** always finds the trivial optimal solution of a star due to the way the algorithm works. In contrast, **KP** can only sparsify 22% of the **Steinlib** instances in comparison to 78% by **ADDJS** and 62% by **BBMRY**.



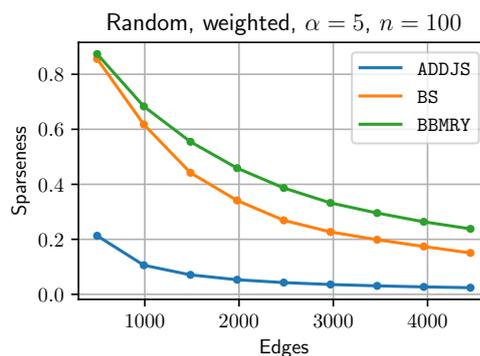
■ **Figure 3** Lightness for **ADDJS**, **BBMRY** and **BS**. **ADDJS** has a maximum lightness of 1.3.



■ **Figure 4** Sparseness comparison averaged on all instances from **Random**.



■ **Figure 5** Sparseness for unweighted graphs with different stretches:  $\alpha = 3$  ( $\triangle$ ),  $\alpha = 5$  ( $+$ ), and  $\alpha = 7$  ( $\times$ ).



■ **Figure 6** Sparseness for weighted graphs. BBMRV and BS have significantly higher sparseness than ADDJS.

For unweighted instances with odd stretch, ADDJS, BS, and EN behave similarly, see Fig. 5. For dense graphs, the sparseness converges and higher stretches unsurprisingly result in lower sparseness. ADDJS generally yields the lowest sparseness, followed by BS and then EN. E.g., over all libraries for  $\alpha = 5$ , BS yields 37% higher sparseness than ADDJS, and EN 47% higher than BS. Although BBMRV's sparseness seems stretch-independent, it is very high compared to all other algorithms. EN shows a higher variance of the sparseness; e.g. for  $\alpha = 3$  its standard deviation is about 4 times that of ADDJS. With higher stretches, the variance decreases.

On weighted graphs, BBMRV and BS yield comparable sparseness; both are, however, significantly worse than ADDJS, see Fig. 6. All algorithms except for ADDJS cannot significantly sparsify the already sparse graphs ( $\varrho_r \leq 10\%$ ) from Steinlib too much: ADDJS's average sparseness is 71.7%, while the sparsenesses of all other algorithms are between 92% and 97%.

**Effective stretch.** For instances with  $n \leq 1000$  ( $n \leq 11000$  for Steinlib), we computed an APSP on the input graphs and the resulting spanners to calculate the effective stretches. More precisely, for each vertex pair  $(u, v)$ , we can compare the length of the shortest path in the input graph to that in the spanner, and obtain an effective stretch specific to  $(u, v)$ . We can aggregate over the effective stretches for all vertex pairs. Table 4 provides an overview of the effective mean, mean maximum, and maximum stretches over all instances. ADDJS gives the overall highest mean stretch and thus always utilizes the limit provided by  $\alpha$ . Interestingly, BBMRV, BS, and EN never use the available stretch of 7 for unweighted graphs. For unweighted complete graphs, EN does not yield a mean stretch over 2 (cf. [15, Appendix, Figure 11]), regardless of  $\alpha$ . KP behaves similar to ADDJS and even yields a higher mean stretch on those instances both ADDJS and KP could solve. BBMRV and BS yield lower mean stretches in comparison to ADDJS for weighted graphs. In summary, these results directly correlate to the previous observations. By having a low mean stretch and not utilizing the allowed stretch, it is clear that some potential for sparsification is not fully exploited, so the lightness and sparseness are also worse.

### 4.3 Further Properties for Weighted Spanners

Finally, we want to touch on the graph structure of spanners particularly on *weighted graphs*. We are interested in their mean degrees, and the number of edges in the shortest paths.

■ **Table 4** Effective mean (mean max, max) stretch for multiple configurations.

	$\alpha$	ADDJS	BS	BBMR	KP/EN
unweighted	2	1.36 (1.98, 2.00)	–	1.10 (1.86, 2.00)	KP: 1.36 (1.87, 2.00)
	3	1.77 (2.94, 3.00)	1.53 (2.77, 3.00)	1.12 (2.06, 3.00)	EN: 1.43 (2.61, 3.00)
	4	2.16 (3.89, 4.00)	–	1.13 (2.13, 4.00)	–
	5	2.47 (4.77, 5.00)	1.77 (3.68, 5.00)	1.13 (2.11, 4.00)	EN: 1.59 (3.13, 5.00)
	7	2.86 (6.43, 7.00)	1.83 (4.16, 6.00)	1.13 (2.12, 5.00)	EN: 1.73 (3.52, 6.00)
weighted	2	1.03 (1.85, 2.00)	–	1.00 (1.19, 2.00)	–
	3	1.07 (2.70, 3.00)	1.01 (1.16, 2.61)	1.00 (1.65, 3.00)	–
	4	1.11 (3.49, 4.00)	–	1.01 (1.99, 4.00)	–
	5	1.16 (4.20, 5.00)	1.01 (1.37, 3.26)	1.01 (2.30, 5.00)	–
	7	1.25 (5.44, 7.00)	1.01 (1.60, 3.49)	1.01 (2.69, 7.00)	–

**Degree.** For  $\alpha = 3$ , BS yields a higher mean degree by 30% than BBMR, but is slightly lower for  $\alpha \geq 5$ ; BS is the only algorithm where a higher stretch results in lower mean degrees. Overall, BBMR and BS yield very high mean degrees (similar to the lightness and sparseness) while ADDJS has mean degrees around 3–5. For the Steinlib instances, ADDJS’s spanners yields a mean degree of (on average) 57% of the mean degree of the input graph. In contrast to this, all other algorithms’ spanners have more than 96% of the original mean degree, see [15, Appendix, Figure 10]. All algorithms show a constant or moderately (linearly) increasing dependency between mean degrees of the original graph and the spanner, except for KP. There, the spanners’ mean degrees first increase together with the original mean degree, but for instances with  $\rho_r \gtrsim 50\%$  the former decrease again. Analogous to the sparseness, the mean degrees become lower than ADDJS’s for large relative densities.

**Hops.** Lastly, we take a look at the number of edges (*hops*) of all shortest paths (again considering the instances described for the effective stretch). Overall, the mean hop difference is always positive, i.e., shortest paths in the spanner on average use more edges than in the original graph. Only 2.27% of all vertex pairs have shortest paths with fewer hops in the spanner than in the original graph; 58.90% retain their hop count, and 38.82% gain at least one hop. ADDJS has a higher average mean hop difference of 3.98 than BS (0.26) and BBMR (0.15). The tendency for all algorithms is to yield a higher hop difference with higher stretch.

## 5 Conclusions and Questions

We conducted the first experimental evaluation of polynomial spanner approximations. We can provide a rough guideline on which algorithm to use. If one has the special case of  $\alpha = 2$  and no edge weights, KP may be used for graphs with  $\rho_r \gtrsim 50$  as long as it can solve the instances in feasible time. In all other cases ADDJS (the oldest and most simplistic approach!) should be the algorithm of choice, as long as the graphs have reasonable sizes. It provides the sparsest and lightest spanners within a reasonable running time. Especially for weighted graphs, no other algorithm can calculate spanners of comparable quality. Only for very large unweighted graphs, BS is a good alternative to ADDJS due to its low running time. BBMR’s subpar performance makes its only sensible for directed graphs, as it is the only algorithm capable of handling such instances. Lastly, while EN can produce good spanners, it is never strictly better in running time nor quality than ADDJS or BS.

Overall, the practical strength of the algorithmic idea of ADDJS seems to be established. Based thereon, it now seems worthwhile to investigate the practical performance of proposed improvements. In particular, this includes the mentioned implementation-wise more intricate

but theoretically faster variants [42, 45] which lose the lightness guarantees. Furthermore, we may ask whether ADDJS can be improved for unweighted graphs by using a specific edge order. Formally, an arbitrary order suffices, but using a BFS or DFS order, or sequentially considering sets of independent edges, may further decrease the spanner size in practice. It may also be interesting to investigate the algorithmic reasons for the significant running time degradation of ADDJS (and BBMR) for  $\alpha = 2$  on unweighted graphs, and to find theoretical results complementing the practical findings on the measures considered in Sect. 4.3.

---

## References

- 1 Reyan Ahmed, Greg Bodwin, Faryad Darabi Sahneh, Keaton Hamm, Mohammad Javad Latifi Jebelli, Stephen Kobourov, and Richard Spence. Graph spanners: A tutorial review. *Computer Science Review*, 37:100253, 2020. doi:10.1016/j.cosrev.2020.100253.
- 2 Reyan Ahmed, Keaton Hamm, Mohammad Javad Latifi Jebelli, Stephen Kobourov, Faryad Darabi Sahneh, and Richard Spence. Approximation algorithms and an integer program for multi-level graph spanners. In *Proc. SEA 2019*, volume 11544 of *LNCS*, pages 541–562. Springer, 2019. doi:10.1007/978-3-030-34029-2.
- 3 Stephen Alstrup, Søren Dahlgaard, Arnold Filtser, Morten Stöckel, and Christian Wulff-Nilsen. Constructing light spanners deterministically in near-linear time. In *Proc. ESA 2019*, volume 144 of *LIPICs*, pages 4:1–4:15. LZI Dagstuhl, 2019. doi:10.4230/LIPICs.ESA.2019.4.
- 4 Ingo Althöfer, Gautam Das, David Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete Computational Geometry*, 9:81–100, 1993. doi:10.1007/BF02189308.
- 5 Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985. doi:10.1145/4221.4227.
- 6 Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures & Algorithms*, 30(4):532–563, 2007. doi:10.1002/rsa.20130.
- 7 Benchmark and experimental data. Url. <https://tcs.uos.de/research/spanner>, 2022.
- 8 Piotr Berman, Arnab Bhattacharyya, Konstantin Makarychev, Sofya Raskhodnikova, and Grigory Yaroslavtsev. Approximation algorithms for spanner problems and directed Steiner forest. *Information and Computation*, 222:93–107, 2013. doi:10.1016/j.ic.2012.10.007.
- 9 Arnab Bhattacharyya, Elena Grigorescu, Kyomin Jung, Sofya Raskhodnikova, and David P. Woodruff. Transitive-closure spanners. In *Proc. SODA 2009*, pages 932–941. ACM SIAM, 2009. doi:10.1137/110826655.
- 10 Quirijn W. Bouts, Alex P. ten Brink, and Kevin Buchin. A framework for computing the greedy spanner. In *Proc. SoCG 2014*, pages 11–19. ACM, 2014. doi:10.1145/2582112.2582154.
- 11 Leizhen Cai. NP-completeness of minimum spanner problems. *Discrete Applied Mathematics*, 48(2):187–194, 1994. doi:10.1016/0166-218X(94)90073-6.
- 12 Paul B. Callahan and S. Rao Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM*, 42(1):67–90, 1995. doi:10.1145/200836.200853.
- 13 Barun Chandra, Gautam Das, Giri Narasimhan, and José Soares. New sparseness results on graph spanners. In *Proc. SoCG 1992*, pages 192–201. ACM, 1992. doi:10.1145/142675.142717.
- 14 Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. The open graph drawing framework (ogdf). In Roberto Tamassia, editor, *Handbook of graph drawing and visualization*, chapter 17. CRC press, 2014.
- 15 Markus Chimani and Finn Stutzenstein. Spanner approximations in practice. *arXiv*, 2022. arXiv:2107.02018.
- 16 Kenneth Lee Clarkson. Approximation algorithms for shortest path motion planning. In *Proc. STOC 1987*, pages 56–65. ACM, 1987. doi:10.1145/28395.28402.

- 17 Lenore J. Cowen. Compact routing with minimum stretch. *Journal of Algorithms*, 38(1):170–183, 2001. doi:10.1006/jagm.2000.1134.
- 18 Lenore J. Cowen and Christopher G. Wagner. Compact roundtrip routing in directed networks. *Journal of Algorithms*, 50(1):79–95, 2004. doi:10.1016/j.jalgor.2003.08.001.
- 19 IBM ILOG Cplex. V12. 1: User’s manual for cplex. *International Business Machines Corporation*, 46(53):157, 2009. URL: <https://www.ibm.com/analytics/cplex-optimizer>.
- 20 Michael Dinitz and Robert Krauthgamer. Directed spanners via flow-based linear programs. *arXiv*, 2010. arXiv:1011.3701.
- 21 Michael Dinitz and Zeyu Zhang. Approximating low-stretch spanners. In *Proc. SODA 2016*, pages 821–840. ACM SIAM, 2016. doi:10.1137/1.9781611974331.ch59.
- 22 Michael Elkin. An unconditional lower bound on the time-approximation trade-off for the distributed minimum spanning tree problem. *SIAM Journal on Computing*, 36(2):433–456, 2006. doi:10.1137/S0097539704441058.
- 23 Michael Elkin and Ofer Neiman. Efficient algorithms for constructing very sparse spanners and emulators. *ACM Transactions on Algorithms*, 15(1), 2018. doi:10.1145/3274651.
- 24 Michael Elkin, Ofer Neiman, and Shay Solomon. Light spanners. *SIAM Journal on Discrete Mathematics*, 29(3):1312–1321, 2015. doi:10.1137/140979538.
- 25 Michael Elkin and David Peleg. Approximating k-spanner problems for  $k > 2$ . *Theoretical Computer Science*, 337(1):249–277, 2005. doi:10.1016/j.tcs.2004.11.022.
- 26 Michael Elkin and Shay Solomon. Fast constructions of lightweight spanners for general graphs. *ACM Transactions on Algorithms*, 12(3), 2016. doi:10.1145/2836167.
- 27 Mohammad Farshi and Joachim Gudmundsson. Experimental study of geometric t-spanners. *ACM Journal on Experimental Algorithmics*, 14, 2010. doi:10.1145/1498698.1564499.
- 28 Giorgio Gallo, Michael D. Grigoriadis, and Robert E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18(1):30–55, 1989. doi:10.1137/0218003.
- 29 Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988. doi:10.1145/48014.61051.
- 30 Andrew Vladislav Goldberg. Finding a maximum density subgraph. Technical Report UCB/CSD-84-171, EECS Department, University of California, Berkeley, 1984.
- 31 Madhav Jha and Sofya Raskhodnikova. Testing and reconstruction of Lipschitz functions with applications to data privacy. *SIAM Journal on Computing*, 42(2):700–731, 2013. doi:10.1137/110840741.
- 32 J. Mark Keil. Approximating the complete Euclidean graph. In *Proc. SWAT 1988*, volume 318 of *LNCS*, pages 208–213. Springer, 1988.
- 33 Thorsten Koch, Alexander Martin, and Stefan Voß. SteinLib: An updated library on Steiner tree problems in graphs. Technical Report ZIB-Report 00-37, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 2000. URL: <http://steinlib.zib.de/steinlib.php>.
- 34 Guy Kortsarz and David Peleg. Generating sparse 2-spanners. *Journal of Algorithms*, 17(2):222–236, 1994. doi:10.1006/jagm.1994.1032.
- 35 Gary L. Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. Improved parallel algorithms for spanners and hopsets. In *Proc. SPAA 2015*, pages 192–201. ACM, 2015. doi:10.1145/2755573.2755574.
- 36 Gary L. Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In *Proc. SPAA 2013*, pages 196–203. ACM, 2013. doi:10.1145/2486159.2486180.
- 37 David Peleg. Proximity-preserving labeling schemes and their applications. In *Proc. WG 1999*, volume 1665 of *LNCS*, pages 30–41. Springer, 1999.
- 38 David Peleg and Alejandro A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989. doi:10.1002/jgt.3190130114.
- 39 David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. In *Proc. PODC 1987*, pages 77–85. ACM, 1987. doi:10.1145/41840.41847.

- 40 David Peleg and Eli Upfal. A trade-off between space and efficiency for routing tables. *Journal of the ACM*, 36(3):510–530, 1989. doi:10.1145/65950.65953.
- 41 Gerhard Reinelt. TspLib—a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991. doi:10.1287/ijoc.3.4.376.
- 42 Liam Roditty and Uri Zwick. On dynamic shortest paths problems. *Algorithmica*, 61:389–401, 2011.
- 43 H. Shpungin and M. Segal. Near optimal multicriteria spanner constructions in wireless ad-hoc networks. In *IEEE INFOCOM 2009*, pages 163–171, 2009. doi:10.1109/INFOCOM.2009.5061918.
- 44 Mikkel Sigurd and Martin Zachariasen. Construction of minimum-weight spanners. In *Proc. ESA 2004*, volume 3221 of *LNCS*, pages 797–808. Springer, 2004.
- 45 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):1–24, 2005. doi:10.1145/1044731.1044732.