2022

# Bounded-degree Plane Geometric Spanners: Connecting the Dots Between Theory and Practice

Matthew Alexander Graham
*University of North Florida*, mattgrahamatc@gmail.com

BOUNDED-DEGREE PLANE GEOMETRIC SPANNERS:
CONNECTING THE DOTS BETWEEN THEORY AND PRACTICE

by

Matthew A. Graham

A thesis submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

April 20, 2022

The thesis "Bounded-degree Plane Geometric Spanners: Connecting the Dots Between Theory and Practice" submitted by Matthew A. Graham in partial fulfillment of the requirements for the degree of Master of Science in Computing and Information Sciences has been

Approved by the thesis committee:                                     Date:

_____                    _____
Dr. Anirban Ghosh
Thesis Advisor and Committee Chairperson


_____                    _____
Dr. Asai Asaithambi
Committee Member


_____                    _____
Dr. Ayan Dutta
Committee Member

# ACKNOWLEDGEMENTS

First, I would like to thank my thesis committee for your time, dedication, and contribution towards my education. Each of you has played a tremendous role in my growth as a computer scientist. Dr. Dutta–your enthusiasm in the Artificial Intelligence class ignited my interest in graph theory enough to make me jump headlong into a thesis in this area. Dr. Asaithambi–your expertise in the classroom is paying off for me today as I design state machines to describe applications for use in the broadcast television industry, and your guidance from the office has also proven instrumental in the completing this degree. Dr. Ghosh–thank you for believing in me when I was just a B.S. in Psychology graduate. Your leadership throughout this project has been pivotal in connecting the dots of this thesis.

Next, I have the utmost gratitude and appreciation for my spanner brothers–Lucas Mougeot, David Wisnosky, and Fred Anderson–whose camaraderie and contributions towards the algorithm implementations and visualizations were legendary. I couldn't have done it without you.

Last, I could not have made it this far without the full love and support of my lovely wife, Amanda, or my two bright, young stars, Jack and Maxeen. This culmination of seven years of study is as much yours to celebrate as it is mine. Your patience and trust has been the foundation of my life.

# CONTENTS

# TABLES

# ALGORITHMS

ABSTRACT

The construction of bounded-degree plane geometric spanners has been a focus of interest since 2002 when Bose, Gudmundsson, and Smid proposed the first algorithm to construct such spanners. To date, eleven algorithms have been designed with various trade-offs in degree and stretch factor. We have implemented these sophisticated algorithms in C++ using the CGAL library and experimented with them using large synthetic and real-world pointsets. Our experiments have revealed their practical behavior and real-world efficacy. We share the implementations via GitHub[1] for broader uses and future research.

We present a simple practical algorithm, named APPXSTRETCHFACTOR, that can estimate stretch factors (obtains lower bounds on the exact stretch factors) of geometric spanners – a challenging problem for which no practical algorithm is known yet. In our experiments with bounded-degree plane geometric spanners, we find that APPXSTRETCHFACTOR estimates stretch factors almost precisely. Further, it gives linear runtime performance in practice for the pointset distributions considered in this work, making it much faster than the naive Dijkstra-based algorithm for calculating stretch factors.

---

[1] `https://github.com/mgatc/geometric-spanners`

# INTRODUCTION

Let $G$ be the complete Euclidean graph on a given set $P$ of $n$ points embedded in the Euclidean plane. A geometric $t$-spanner on $P$ is a geometric graph $G' := (P, E)$, a subgraph of $G$ such that for every pair of points $u, v \in P$, the distance between them in $G'$ (the Euclidean length of a shortest path between $u, v$ in $G'$) is at most $t$ times their Euclidean distance $|uv|$, for some $t \geq 1$. It is easy to check that $G$ itself is a 1-spanner with $\Theta(n^2)$ edges. The quantity $t$ is referred to as the stretch factor of $G'$. If there is no need to specify $t$, we simply use the term geometric spanner and assume that there exists some $t$ for $G'$. We say that $G'$ is plane if it is crossing-free. $G'$ is degree-$k$ or is said to have bounded-degree if its degree is at most $k$. In this work, we experiment with bounded-degree plane geometric spanners. See Figure 1.1 for an example of such a spanner.

Bounded-degree plane geometric spanners have been an area of interest in computational geometry for a long time. Non-crossing edges make them suitable for wireless network applications where edge crossings create communication interference. The absence of crossing edges also makes them useful for the design of road networks to eliminate high-budget flyovers. Such spanners have $O(n)$ edges (at most $3n - 6$ edges); as a result, they are less expensive to store and navigate. Further, shortest-path algorithms run quicker on them since they are sparse. Bounded-degree helps in reducing on-site equipment costs.

Bose, Gudmundsson, and Smid [13] were the first to show that there always exists a plane geometric $\sigma(\pi + 1)$-spanner of degree at most 27 on any pointset, where $\sigma$ denotes an upper
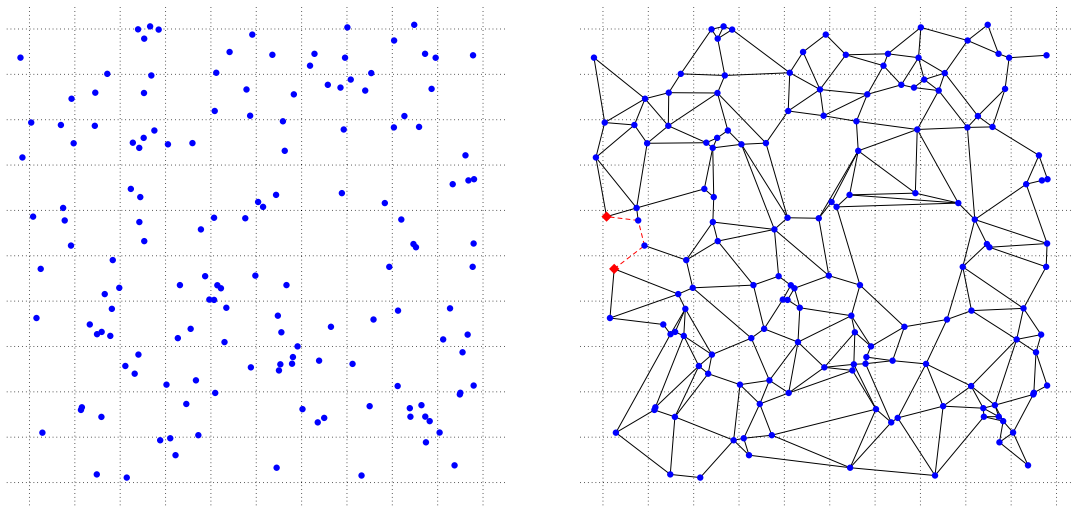
Figure 1.1: Left: A set $P$ of 150 points, generated randomly within a square. Right: A plane degree-6 spanner on $P$ with stretch factor $\approx 1.82$. The pair of points for which the spanner achieves a stretch factor of $\approx 1.82$ is shown in red along with the shortest path between them.

bound for the stretch factor of $L_2$-Delaunay triangulations[1] (the current best known value is $\sigma = 1.998$ due to Xia [44]). This result was subsequently improved in a long series of papers [9, 12, 14, 16, 33, 34, 36] in terms of degree and/or stretch factor. Bonichon et al. [11] reduced the degree to 4 with $t \approx 156.8$. Soon after this, Kanj et al. improved this stretch factor upper bound to 20 in [32]. A summary of these results is presented in Table 1.1. This family of spanner construction algorithms has turned out to be a fascinating application of the Delaunay triangulation. Note that all these algorithms produce bounded-degree plane subgraphs of the complete Euclidean graph on $P$ with constant stretch factors.

The intriguing question that remains to be answered is whether the degree can be reduced to 3 while keeping $t$ bounded; refer to [15, Problem 14] and [42, Chapter 32]. Interestingly, if one does not insist on constructing a *plane* spanner, Das and Heffernan [23] showed that degree 3 is always achievable. Narasimhan and Smid [39, Section 20.1] show that no degree-2 plane spanner of the infinite integer lattice can have a constant stretch factor. Thus, a minimum

---

[1]A triangulation $T$ for a pointset $P$ is referred to as a $L_2$-Delaunay triangulation if no point in $P$ lies inside the circumcircle of any triangle in $T$.

Table 1.1: A summary of results on constructions of bounded degree plane geometric spanners, sorted by the degree they guarantee. The best known upper bound of $\sigma = 1.998$ for the stretch factor of the $L_2$-Delaunay triangulation [44] is used in this table for expressing the stretch factors.

| Reference | Degree | Stretch Factor |
|---|---|---|
| Bose, Gudmundsson, and Smid [13] | 27 | $\sigma(\pi + 1) \approx 8.3$ |
| Li and Wang [36] | 23 | $\sigma(1 + \frac{\pi}{\sqrt{2}}) \approx 6.4$ |
| Bose, Smid, and Xu [16] | 17 | $\sigma(2 + 2\sqrt{3} + \frac{3\pi}{2} + 2\pi \sin\frac{\pi}{12}) \approx 23.6$ |
| Kanj, Perković, and Xia [33] | 14 | $\sigma(1 + \frac{2\pi}{14\cos(\pi/14)}) \approx 2.9$ |
| Kanj and Xia [34] | 11 | $\sigma(\frac{2\sin(2\pi/5)\cos(\pi/5)}{2\sin(2\pi/5)\cos(\pi/5)-1}) \approx 5.7$ |
| Bose, Hill, and Smid [14] | 8 | $\sigma\left(1 + \frac{2\pi}{6\cos(\pi/6)}\right) \approx 4.4$ |
| Bose, Carmi, and Chaitman-Yerushalmi [12] | 7 | $\sigma(1 + \sqrt{2})^2 \approx 11.6$ |
| Bose, Carmi, and Chaitman-Yerushalmi [12] | 6 | $\sigma\left(\frac{1}{1-\tan(\pi/7)(1+1/\cos(\pi/14))}\right) \approx 81.7$ |
| Bonichon, Gavoille, Hanusse, and Perković [9] | 6 | 6 |
| Bonichon, Kanj, Perković, and Xia [11] | 4 | $\sqrt{4 + 2\sqrt{2}}(19 + 29\sqrt{2}) \approx 156.8$ |
| Kanj, Perković, and Türkoğlu [32] | 4 | 20 |

degree of 3 is necessary to achieve a constant stretch factor. If $P$ is convex, then it is always possible to construct a degree-3 plane geometric spanners, see [4, 7, 32]. From the other direction, lower bounds on the stretch factors of plane spanners for finite pointsets have been investigated in [24, 25, 35, 37]. In-browser visualizations of some of the algorithms (those based on the $L_2$-Delaunay triangulation) have been recently presented in [3]. In related works, the construction of plane hop spanners (where the number of hops in shortest paths is of interest) for unit disk graphs has been considered in [6, 19, 26].

The most notable experimental work for geometric spanners is done by Farshi and Gudmundsson [27]. The authors engineered and experimented with some of the well-known geometric spanners construction algorithms published before 2009. However, the authors did not use the algorithms considered in this work in their experiments. Planarity and bounded-degree are important concerns in geometric network design. Hence, we found it motivating to implement the eleven algorithms (refer to Table 1.1) meant to construct bounded-degree plane geometric spanners. Asymptotic runtimes and various theoretical

bounds do not always do justice in explaining the real-world performance of algorithms, especially in computational geometry, because of computationally heavy operations needed for various geometric calculations. Experiments reveal their real-world performance. A unique aspect of the family of bounded-degree plane spanner construction algorithms is that users cannot specify an arbitrary value of $t$ and/or degree for spanner construction. It is a deviation from many standard spanner algorithms; see [12,39] for a review of such algorithms. This makes experiments with them even more interesting.

**Our contributions.** (i) We experimentally compare the aforementioned eleven bounded-degree plane spanner construction algorithms by implementing them carefully in C++ using the popular CGAL library [41] and running them on large synthetic and real-world pointsets. The largest pointset contains 1.9 million points approximately. For broader uses of these sophisticated algorithms, we share the C++ implementations via GitHub[2]. The comparisons are performed based on their runtimes and degree, stretch factor, and lightness of the generated spanners. We present a brief overview of the algorithms implemented and our experimental results in Sections 2 and 4, respectively.

(ii) In doing experiments with spanners, we find that stretch factor measurement turns out to be a severe bottleneck when $n$ is large. To address this, we have designed a new fast algorithm named APPXSTRETCHFACTOR that can estimate the stretch factor of a given spanner (not necessarily plane). In our experiments, we find that it can estimate stretch factors with high accuracy for the class of geometric spanners dealt with in this work. It is considerably faster than the naive Dijkstra-based exact stretch factor measurement algorithm in practice. To our knowledge, no such practical algorithm exists in the literature. Further, it can be easily parallelized, making it very useful for estimating stretch factors of large spanners. See Section 3 for a description of this algorithm.

---

[2]https://github.com/mgatc/geometric-spanners

# CHAPTER 2

## Algorithms implemented

Every algorithm designed to date for constructing bounded-degree plane geometric spanners relies on some variant of Delaunay triangulation. The rationale behind this is that such triangulations are geometric spanners [10, 21, 22, 44] and are plane by definition. As a result, the family of plane spanner construction algorithms considered in this work has turned out to be a fascinating application of Delaunay triangulation. It is essential to know that Delaunay triangulations have unbounded degrees and cannot be used as bounded-degree plane spanners.

In this section, we provide a brief description for each of the eleven algorithms considered in this work. Appropriate abbreviations using the authors' names and dates of publication are used for naming purposes. Since most of these algorithms are involved, so we urge the reader to refer to the original papers for a deeper understanding and correctness proofs. For visualizing some of these algorithms, we recommend the interactive in-browser applet[1] developed by us; refer to [3]. To observe variations in spanner construction between the algorithms see Appendix A.

---

[1]`https://ghoshanirban.github.io/bounded-degree-plane-spanners/index.html`

---

**Algorithm 1:** `ReverseLowDegreeOrdering(`$DT$`)`

---
**1** Allocate $\Phi[1 \ldots n]$;
**2** Make a copy of $DT$ and call it $H$;
**3 for** $i = 1$ **to** $n$ **do**
**4**     Let $u$ be a vertex in $H$ with minimal degree;
**5**     $\Phi[u] \leftarrow n - i + 1$;
**6**     Remove $u$ and all incident edges from $H$;
**7 return** $\Phi$;

---

---

**Algorithm 2:** LW04($P$, $0 < \alpha \leq \pi/2$)

---

**1** $DT \leftarrow L_2\text{-DelaunayTriangulation}(P)$;

**2** $\Phi[1 \ldots n] \leftarrow \text{ReverseLowDegreeOrdering}(DT)$;

**3** $E \leftarrow \emptyset$;

**4** **foreach** $\underline{u \in \Phi}$ **do**

**5**      Divide the area surrounding $u$ into sectors delineated by $u$'s already `processed` neighbors in $DT$;

**6**      Divide each sector into a minimum number of cones $C_u^0, C_u^1, \ldots$ with angle at most $\alpha$;

**7**      **foreach** $\underline{C_u^i}$ **do**

**8**          Let $v_1, v_2, ..., v_m$ be the clockwise-ordered Delaunay neighbors of $u$ in $C_u^i$;

**9**          Add an edge between $u$ and the closest of such neighbors to $E$;

**10**          Add all edges $\{v_j, v_{j+1}\}$ such that $1 \leq j < m$ to $E$;

**11**      Mark $u$ as `processed`;

**12** **return** $E$;

---

In these algorithms, the surrounding of every input point is frequently divided into multiple cones (depending on the algorithm) for carefully selecting edges from the Delaunay triangulation used as the starting point. In our pseudocodes, the cone $i$ of point $u$, considered clockwise, is denoted by $C_i^u$. A triangulation $T$ of a pointset $P$ is said to be a $L_2$-Delaunay triangulation of $P$ if no point in $P$ lies inside the circumcircle of any triangle in $T$. Eight of the eleven algorithms use $L_2$-Delaunay triangulation as the starting point. The remaining three use $L_\infty$ or $TD$-Delaunay triangulations, as described later in this section. In the following, $n$ denotes the size of the input pointset.

- `BGS05`: **Bose, Gudmundsson, and Smid** [13]. This was the first algorithm that can construct bounded-degree plane spanners using the classic $L_2$-Delaunay triangulation. First, a Delaunay triangulation $DT$ of $P$ is constructed. Next, a degree-3 spanning subgraph $SG$ of $DT$ is computed that contains the convex hull of $P$ and is a (possibly degenerate) simple polygon with $P$ as its vertex set. The polygon is then transformed into a simple non-degenerate polygon $Q$. The vertices of $Q$, are processed in an order that is obtained from a breadth-first order of the Delaunay triangulation of $Q$, and then Delaunay edges are carefully added to the resulting graph denoted $G'$, a plane spanner for the vertices of $Q$. Refer to Algorithm 11 for a pseudocode of this algorithm. The authors show that their

algorithm generates degree-27 plane spanners with a stretch factor of $1.998(\pi + 1) \approx 8.3$ and runs in $O(n \log n)$ time.

- LW04: **Li and Wang** [36]. This algorithm is inspired from BSG2005 but is a lot simpler and avoids the use of intermediate (possibly degenerate) polygons. The algorithm computes a reverse low degree ordering of the vertices of the $L_2$-Delaunay triangulation $DT$ constructed on $P$. Then it sequentially considers the vertices in this ordering, divides the surrounding of every such vertex into multiple cones, and then adds short edges from $DT$ to preserve planarity. Refer to Algorithm 2 for a pseudocode of this algorithm. The authors have shown that this algorithm generates degree-23 plane spanners (when the input parameter $\alpha$ of this algorithm is set to $\pi/2$) having a stretch factor of $1.998 \left(1 + \pi/\sqrt{2}\right) \approx 6.4$ and runs in $O(n \log n)$ time.

---

**Algorithm 3:** BSX09$(P,\ 0 < \alpha \le 2\pi/3)$

**1**   $DT \leftarrow L_2\text{-DelaunayTriangulation}(P)$;
**2**   $\Phi[1 \dots n] \leftarrow \text{ReverseLowDegreeOrdering}(DT)$ (use Algorithm 1);
**3**   $E \leftarrow \emptyset$;
**4**   **foreach** $\underline{u \in \Phi}$ **do**
**5**     **if** $\underline{u \text{ has unprocessed Delaunay neighbors}}$ **then**
**6**       Let $v_{closest}$ be the closest unprocessed neighbor to $u$;
**7**       Add the edge $\{u, v_{closest}\}$ to $E$;
**8**       Divide the area surrounding $u$ into a minimum number of cones $C_u^0, C_u^1, \ldots$ with angle at most $\alpha$, such that $v_{closest}$ is on the boundary between the first and last cones;
**9**       **foreach** $\underline{C_u^i \text{ except the first and last}}$ **do**
**10**         **if** $\underline{u \text{ has unprocessed neighbors in } C_u^i}$ **then**
**11**           Let $w$ be the closest unprocessed neighbor to $u$ in the cone;
**12**           Add edge $\{u, w\}$ to $E$;
**13**         **end**
**14**         Let $v_0, v_1, ..., v_{m-1}$ be the clockwise-ordered neighbors of $u$;
**15**         Add all edges $\{v_j, v_{(j+1) \bmod m}\}$ to $E$ such that $0 \le j < m$ and $v_j, v_{(j+1) \bmod m}$ are unprocessed;
**16**       **end**
**17**     **end**
**18**     Mark $u$ as processed;
**19**   **end**
**20**   **return** $E$;

---

- BSX09: **Bose, Smid, and Xu** [16]. This algorithm is quite similar to LW04 in design and also relies on reverse low-degree ordering of the vertices of the Delaunay triangulation. Refer to Algorithm 3. The authors have generalized their algorithm so that it can construct bounded-degree plane spanners from any triangulation of $P$, not necessarily just the $L_2$-Delaunay triangulation (although the $L_2$-Delaunay triangulation is of primary interest to us). When the $L_2$-Delaunay triangulation is used and the parameter $\alpha$ is set to $2\pi/3$, the algorithm generates degree-17 plane spanners having a stretch factor of $\sigma(2 + 2\sqrt{3} + \frac{3\pi}{2} + 2\pi \sin \frac{\pi}{12}) \approx 23.6$ in $O(n \log n)$ time. After computing the triangulation and the reverse low-degree ordering, at every vertex $u$, $\delta = \lceil 2\pi/\alpha \rceil$ Yao cones are initialized such that the closest unprocessed triangulation neighbor falls on a cone boundary and occupies both cones as the *short* edge, which is added to the spanner. In the remaining cones, the closest unprocessed neighbor of $u$ in each cone is added. In all cones, special edges between pairs of neighbors of $u$ are added to the spanner if both the neighbors are unprocessed.

---

**Algorithm 4:** KPX10($P$, integer $k \geq 14$)

1  $DT \leftarrow L_2$-DelaunayTriangulation($P$);
2  **foreach** <u>vertex $u \in DT$</u> **do**
3      Partition the area surrounding $u$ into $k$ disjoint cones of angle $2\pi/k$;
4      In each nonempty cone, **select** the shortest edge in $DT$ incident to $u$;
5      **foreach** <u>maximal sequence of $\ell \geq 1$ consecutive empty cones</u> **do**
6          **if** $\ell > 1$ **then**
7              **select** the first $\lfloor \ell/2 \rfloor$ **unselected** incident $DT$ edges on $u$ clockwise from the sequence of empty cones and the first $\lceil \ell/2 \rceil$ **unselected** $DT$ edges incident on $u$ counterclockwise from the sequence of empty cones;
8          **else**
9              let $ux$ and $uy$ be the incident $DT$ edges on $m$ clockwise and counterclockwise, respectively, from the empty cone;
10             **if** either $ux$ or $uy$ is selected **then select** the other edge (in case it has not been selected); otherwise **select** the shorter edge between $ux$ and $uy$ breaking ties arbitrarily;
11         **end**
12     **end**
13 **end**
14 **return** the $DT$ edges **selected** by both endpoints;

---

- BGHP10: **Bonichon, Gavoille, Hanusse, and Perković** [9]. It was the first algorithm that deviated from the use of $L_2$-Delaunay triangulation; instead, it used $TD$-Delaunay triangulation to select spanner edges, introduced by Chew back in 1989 [22]. For such triangulations, empty equilateral triangles are used for characterization instead of empty circles, as needed in the case of $L_2$-Delaunay triangulations. $TD$-Delaunay triangulations are plane 2-spanners but may have an unbounded degree. BGHP10 first extracts a degree-9 subgraph from the $TD$-Delaunay triangulation that has a stretch factor of 6. Then using some local modifications, the degree is reduced from 9 to 6 but the stretch factor remains unchanged. Refer to Algorithm 16. It uses internally Algorithms 17 - 23. In this algorithm, each edge incident to a node is charged to some cone of that node. The algorithm runs in $O(n \log n)$ time, as shown by the authors.

---

**Algorithm 5:** BCC12$(P, \Delta \in \{6,7\})$

1   $DT \leftarrow L_2$-DelaunayTriangulation$(P)$;
2   $E, E^* \leftarrow \emptyset$;
3   Initialize $k = \Delta + 1$ cones surrounding each vertex $u$, oriented such that the shortest edge incident on $u$ falls on a boundary;
4   **foreach** $\{u, v\} \in DT$ in order of nondecreasing length **do**
5      **if** $\forall C_u^i$ containing $\{u,v\}$, $C_u^i \cap E = \emptyset$ and $\forall C_v^j$ containing $\{u,v\}$, $C_v^j \cap E = \emptyset$ **then**
6         Add edge $\{u,v\}$ to $E$;
7      **end**
8   **end**
9   **foreach** $\{u,v\} \in E$ **do**
10     Wedge$_\Delta(u, v)$;
11     Wedge$_\Delta(v, u)$;
12   **end**
13   **return** $E \cup E^*$;

---

- KPX10: **Kanj, Perković, and Xia** [33]. For every vertex $u$ in the $L_2$-Delaunay triangulation, its surrounding is divided into $k \geq 14$ cones. In every nonempty cone of $u$, the shortest Delaunay edge incident on $u$ is selected. After this, a few additional Delaunay edges are also selected using some criteria based on cone sequences. See Algorithm 4 for a complete description of this algorithm with the technical details. When $k$ is set to 14, degree-14 plane spanners are generated having a stretch factor of $1.998 \left(1 + \frac{2\pi}{14 \cos(\pi/14)}\right) \approx 2.9$. Note

that out of the 11 algorithms we have implemented in this work, this algorithm gives the best theoretical guarantee on the stretch factor; see Table 1.1. KPX10 runs in $O(n \log n)$ time.

---

**Algorithm 6:** $\mathtt{Wedge}_7(u, v_i)$

---

**1** **foreach** $C_u^z$ containing $\{u, v_i\}$ **do**
**2** $\quad$ Let $\{u, v_j\}$ and $\{u, v_k\}$ be the first and last edges of $DT$ in the cone;
**3** $\quad$ Add all edges $\{v_m, v_{m+1}\}$ to $E^*$ such that $j < m < i - 1$ or $i < m < k - 1$;
**4** $\quad$ **if** $\{u, v_{i+1}\} \in C_u^z$ and $v_{i+1} \neq v_k$ and $\angle uv_iv_{i+1} > \pi/2$ **then**
**5** $\quad\quad$ | Add edge $\{v_i, v_{i+1}\}$ to $E^*$;
**6** $\quad$ **end**
**7** $\quad$ **if** $\{u, v_{i-1}\} \in C_u^z$ and $v_{i-1} \neq v_k$ and $\angle uv_iv_{i-1} > \pi/2$ **then**
**8** $\quad\quad$ | Add edge $\{v_i, v_{i-1}\}$ to $E^*$;
**9** $\quad$ **end**
**10** **end**

---

**Algorithm 7:** $\mathtt{KX12}(P)$

---

**1** $DT \leftarrow L_2\text{-}\mathtt{DelaunayTriangulation}(P)$;
**2** **foreach** vertex $u \in DT$ **do**
**3** $\quad$ In each $\mathtt{wide}$ sequence (a sequence of exactly three consecutive edges incident to a vertex whose overall angle is at least $4\pi/5$) around $u$, $\mathtt{select}$ the edges of the sequence;
**4** $\quad$ Partition the remaining space surrounding $u$ not in a $\mathtt{wide}$ sequence into a minimum number of disjoint cones of maximum angle $\pi/5$;
**5** $\quad$ In each nonempty cone, $\mathtt{select}$ the shortest edge incident to $u$;
**6** $\quad$ In each empty cone, let $ux$ and $uy$ be the incident $DT$ edges on $u$ clockwise and counterclockwise, respectively, from the empty cone;
**7** $\quad$ If either $ux$ or $uy$ is selected then $\mathtt{select}$ the other edge (in case it has not been selected); otherwise $\mathtt{select}$ the longer edge between $ux$ and $uy$ breaking ties arbitrarily;
**8** **end**
**9** **return** all edges selected by both incident vertices;

---

- KX12: **Kanj and Xia** [34]. This $O(n \log n)$-time algorithm takes a different approach in contrast with the previous ones, although it still uses the $L_2$-Delaunay triangulation $DT$ as the starting point. Every vertex $u$ in $DT$ selects at most 11 of its incident edges in $DT$, and edges that are selected by both endpoints are kept. As such, it is guaranteed that the degree of the resulting subgraph is at most 11. The stretch factors of the generated spanners is shown to be at most $1.998 \left( \frac{2 \sin(2\pi/5) \cos(\pi/5)}{2 \sin(2\pi/5) \cos(\pi/5) - 1} \right) \approx 5.7$. Refer to Algorithm 7.

---

**Algorithm 8:** BHS18($P$)

---

1  $DT \leftarrow L_2$-DelaunayTriangulation($P$);
2  Let $m$ be the number of edges in $DT$;
3  $L$ be the edges $\in DT$ sorted in non-decreasing order of bisector-distance;
4  $E_A \leftarrow$ addIncident($L$), $E_{CAN} \leftarrow \emptyset$;
5  **foreach** $\{u,v\} \in E_A$ **do**
6  $\quad E_{CAN} \leftarrow E_{CAN} \cup$ addCanonical($u,v$) $\cup$ addCanonical($v,u$);
7  **end**
8  **return** $E_A \cup E_{CAN}$;

---

---

**Algorithm 9:** addIncident($L$)

---

1  $E_A \leftarrow \emptyset$;
2  **foreach** $\{u,v\} \in L$ **do**
3  $\quad$ Let $i$ be the cone of $u$ containing $v$;
4  $\quad$ **if** $\{u,w\} \notin E_A$ for all $w \in N_i^u \wedge \{v,y\} \notin E_A$ for all $y \in N_{i+3}^v$ **then**
5  $\quad\quad$ Add $\{u,v\}$ to $E_A$;
6  $\quad$ **end**
7  **end**
8  **return** $E_A$;

---

- BCC12-7, BCC12-6: **Bose, Carmi, and Chaitman-Yerushalmi** [12]. The authors present two algorithms in their paper. Whereas previous algorithms used strategies involving iterating over the vertices one-by-one, this algorithm takes the approach of iterating over the edges of the Delaunay triangulation in order of non-decreasing length to query agreement among the vertices for bounding degrees. BCC12-7, the simpler of the two, produces $1.998(1 + \sqrt{2})^2 \approx 11.6$-spanners with degree 7. BCC12-6, on the other hand, constructs $11.998 \left( \frac{1}{1-\tan(\pi/7)(1+1/\cos(\pi/14))} \right) \approx 81.7$-spanners with degree 6 but not all edges come from the $L_2$-Delaunay triangulation. Both these algorithms run in $O(n \log n)$ time. See Algorithm 5. The parameter $\Delta \in \{7, 6\}$ is used to control the degree. Depending on $\Delta$, either Algorithm 6 or Algorithm 33 is invoked.

- BKPX15: **Bonichon, Kanj, Perković, and Xia** [11]. This algorithm uses the $L_\infty$-Delaunay triangulation and was the first degree-4 algorithm. For such triangulations, empty axis parallel squares are used for characterization instead of empty circles, as needed in the case of $L_2$-Delaunay triangulations. The $L_\infty$-distance between two points $u, w$ is

---

**Algorithm 10:** addCanonical($u, v$)

**1** $E' \leftarrow \emptyset$;

**2** Let $i$ be the cone of $u$ containing $v$;

**3** Let $e_{first}$ and $e_{last}$ be the first and last canonical edge in $Can_i^{\{u,v\}}$;

**4** **if** $\underline{Can_i^{\{u,v\}}}$ has at least 3 edges **then**

**5**      **foreach** $\underline{\{s,t\} \in Can_i^{\{u,v\}} \setminus \{e_{first}, e_{last}\}}$ **do**

**6**          |   Add $\{s,t\}$ to $E'$;

**7**      **end**

**8** **end**

**9** **if** $\underline{v \in \{e_{first}, e_{last}\}}$ **and** there is more than one edge in $Can_i^{\{u,v\}}$ **then**

**10**      Add the edge of $Can_i^{\{u,v\}}$ incident to $v$ to $E'$;

**11** **end**

**12** **foreach** $\underline{\{y,z\} \in \{e_{first}, e_{last}\}}$ **do**

**13**      **if** $\underline{\{y,z\} \in N_{i-1}^z}$ **then**

**14**          |   Add $\{y,z\}$ to $E'$;

**15**      **end**

**16**      **if** $\underline{\{y,z\} \in N_{i-2}^z}$ **then**

**17**          **if** $\underline{N_{i-2}^z \cap E_A}$ does not have an edge incident to $z$ **then**

**18**              |   Add $\{y,z\}$ to $E'$;

**19**          **end**

**20**          **if** $\underline{N_{i-2}^z \cap E_A \setminus \{y,z\}}$ has an edge incident to $z$ **then**

**21**              |   Let $\{w,y\}$ be the canonical edge of $z$ incident to $y$;

**22**              |   Add $\{w,y\}$ to $E'$;

**23**          **end**

**24**      **end**

**25** **end**

**26** **return** $\underline{E'}$;

---

defined as $d_\infty(u,w) = \max(d_x(u,w), d_y(u,w))$. From the $L_\infty$-Delaunay triangulation, a directed $L_\infty$-distance-based Yao graph $\overrightarrow{Y_4^\infty}$ is constructed that is a plane $\sqrt{20 + 14\sqrt{2}}$-spanner. Then a degree-8 subgraph $H_8$ of $Y_4^\infty$ is constructed. Finally, some redundant edges are removed and new shortcut edges are added to obtain the final plane degree-4 spanner with a stretch factor of $\sqrt{20 + 14\sqrt{2}}(19 + 29\sqrt{2}) \approx 156.8$. No runtime analysis is presented by the authors. Refer to Algorithm 24. The algorithm divides the space around each point into 4 cones, separated by the $x$ and $y$-axes after translating the point to the origin. Each cone has an associated `charge`, which can be $0, 1,$ or $2$. The algorithm labels certain edges as follows. Each edge will be an `anchor` or a `non-anchor` and `weak` or `strong`. Further, each edge may have an additional label of `start-of-odd-chain-anchor`. A `weak`

`anchor chain` is a path $w_0, w_1, w_2, ...w_k$ of maximal length consisting of `weak anchors` such that the cone of each edge (wrt the source vertex) alternates between some $i$ and $i+2$. `Canonical` edges are edges between consecutive vertices in the ordered neighborhood of a vertex $u$ in a common cone $i$. An edge $(u, v)$ is said to be `dual` if there are two or more edges of $\overrightarrow{Y_4^\infty}$ incident to cone $i$ of $u$ and cone $i+2$ of $v$.

- `KPT17`: **Kanj, Perković, and Türkoğlu** [32]. Akin to `BGHP10`, this algorithm uses the $TD$-Delaunay triangulation and $\Theta$-graph to introduce fresh techniques in spanner construction. Refer to Algorithm 32 for a pseudocode of this algorithm. The authors show that their algorithm generates degree-4 spanners with a stretch factor of 20 and runs in $O(n \log n)$ time. In the following, we define the notations used in the pseudocode.

  - For each vertex, the shortest edge in each odd cone is called an anchor.

  - Cones 1 and 4 are labelled as `blue` and the rest as `white`.

  - The first and last edges incident upon a vertex $u$ in a cone $i$ are called the `boundary` edges of $u$ in $i$.

  - The `canonical` path is made up of all `canonical` edges incident on $u$ in cone $i$, forming a path from one `boundary` edge in the cone to the other.

- `BHS18`: **Bose, Hill, and Smid** [14]. This algorithm produces a plane degree-8 spanner with stretch factor at most $1.998 \left(1 + \frac{2\pi}{6\cos(\pi/6)}\right) \approx 4.4$ using the $L_2$-Delaunay triangulation and $\Theta$-graph. However, the authors do not present any runtime analysis of their algorithm. In `BHS18`, the space around every point $p$ is divided into six cones and are oriented such that a boundary lies on the $x$-axis after translating $p$ to the origin. The algorithm starts with the $L_2$-Delaunay triangulation $DT$, then, in order of non-decreasing bisector distance, each edge is added to the spanner if the cones containing it are both empty. For each edge added here, certain *canonical* edges will also be carefully added to the spanner. Refer to Algorithm 8. In the following, we define the notations used in this pseudocode.

- The `bisector-distance` $[pq]$ between $p$ and $q$ is the distance from $p$ to the orthogonal projection of $q$ onto the bisector of $C_i^p$ where $q \in C_i^p$.

- Let $\{q_0, q_1, ..., q_{d-1}\}$ be the sequence of all neighbors of $p$ in $DT$ in consecutive clockwise order. The neighborhood $N_p$ with apex $p$ is the graph with the vertex set $\{p, q_0, q_1, ..., q_{d-1}\}$ and the edge set $\{\{q_j, q_{j+1}\}\} \cup \{\{q_j, q_{j+1}\}\}, 0 \leq j \leq d-1$, with all values mod $d$. The edges $\{\{q_j, q_{j+1}\}\}$ are called *canonical edges*.

- $N_i^p$ is the subgraph of $N_p$ induced by all the vertices of $N_p$ in $C_i^p$, including $p$.

- Let $Can_i^{\{p,r\}}$ be the subgraph of $DT$ consisting of the ordered subsequence of canonical edges $\{s, t\}$ of $N_i^p$ in clockwise order around apex $p$ such that $[ps] \geq [pr]$ and $[pt] \geq [pr]$.

---

**Algorithm 11:** BGS05$(P)$

---

1   $DT \leftarrow L_2\text{-DelaunayTriangulation}(P)$;
2   $SG \leftarrow \text{SpanningGraph}(DT)$;
3   $Q \leftarrow \text{TransformPolygon}(SG, DT)$;
4   $G' \leftarrow \text{PolygonSpanner}(Q, SG)$;
5   **return** $G'$;

---

**Algorithm 12:** SpanningGraph$(DT)$

---

1   $\Phi[1, \ldots, n] \leftarrow \text{CanonicalOrdering}(DT)$;
2   $SG \leftarrow \emptyset$;
3   Add edges between $v_1, v_2, v_3 \in \Phi$ to $SG$ and mark the vertices as `done`;
4   **for** $v_i \in \Phi \setminus \{v_1, v_2, v_3\}$ **do**
5      Let $u_1, ..., u_k$ be the vertices neighboring $v_i$ in $DT$ marked as `done`;
6      Remove edge $\{u_1, u_2\}$ from $SG$;
7      Add edges $\{v_i, u_1\}$ and $\{v_i, u_2\}$ to $SG$;
8      **if** $k > 2$ **then**
9          Remove edge $\{u_{k-1}, u_k\}$ from $SG$;
10         Add edge $\{v_i, u_k\}$ to $SG$;
11      **end**
12   **end**
13   **return** $SG$;

---

**Algorithm 13:** CanonicalOrdering($DT$)

---

**1** Allocate $\Phi[1, \ldots, n]$;

**2** Make a copy of $DT$ and call it $H$;

**3** Let `reserved` be a set of two consecutive vertices $v_1, v_2$ on the convex hull of $H$;

**4** $\Phi_1 \leftarrow v_1, \Phi_2 \leftarrow v_2$;

**5** **for** $i = 1$ **to** $n - 2$ **do**

**6**      Let $u$ be a vertex of the outer face of $H \setminus$ `reserved` that is adjacent to at most two other vertices on the outer face;

**7**      $\Phi_u \leftarrow n - i + 1$;

**8**      Remove $u$ and all incident edges from $H$;

**9** **end**

**10** **return** $\Phi$;

---

---

**Algorithm 14:** TransformPolygon($SG, DT$)

---

**1** $V \leftarrow \emptyset, E \leftarrow \emptyset$;

**2** Let $s_1, v_1$ be two consecutive vertices on the convex hull of $SG$ in counterclockwise order;

**3** $v_{prev} \leftarrow s_1, v_i \leftarrow v_1$;

**4** Add $v_{prev}$ to $V$;

**5** **do**

**6**      Add $v_i$ to $V$;

**7**      Add $\{v_i, v_{prev}\}$ to $E$;

**8**      Let $v_{next}$ be the neighbor of $v_i \in SG$ such that $v_{next}$ is the next neighbor clockwise from $v_{prev}$;

**9**      $v_{prev} \leftarrow v_i, v_i \leftarrow v_{next}$;

**10** **while** $v_{prev} \neq s_1$ **and** $v_i \neq v_1$;

**11** $E = E \cup \{\{v_i, v_{prev}\}\} \cup DT \setminus SG$;

**12** **return** $(V, E)$;

---

---

**Algorithm 15:** PolygonSpanner($Q, SG$)

---

**1** Let $V, E$ be the vertices and edges of $Q$, respectively;

**2** Let $\rho[1, \ldots, n]$ be the breadth-first ordering of $V$ in $Q$, starting at any vertex in $V$;

**3** $E' \leftarrow SG$;

**4** **foreach** $u \in \rho$ **do**

**5**      Let $s_1, s_2, \ldots, s_m$ be the clockwise ordered neighbors of $u$ in $Q$;

**6**      $s_j, s_k \leftarrow s_m$;

**7**      **if** $u \neq \rho_1$ **then**

**8**          Set $s_j$ and $s_k$ to the first and last vertex in the ordered neighborhood of $u$ where $s_j, s_k \in E'$;

**9**      **end**

**10**      Divide $\angle s_1 u s_j$ and $\angle s_k u s_m$ into an minimum number of cones with maximum angle $\pi/2$;

**11**      In each cone, add the shortest edge in $E$ incident upon $u$ to $E'$ and all edges $\{s_\ell, s_{\ell+1}\}$ such that $1 \leq \ell < j$ **or** $k \leq \ell < m$;

**12** **end**

**13** **return** $E'$;

---

## Algorithm 16: BGHP10($P$)

**1** $DT \leftarrow$ TD-DelaunayTriangulation($P$);

**2** $E \leftarrow \emptyset$;

**3** **foreach** <u>nonempty cone $i$ of vertex $u \in DT$ where $i \in \{1, 3, 5\}$</u> **do**

**4**      Add edge $\{u, \texttt{closest}(u, i)\}$ to $E$;

**5**      $\texttt{charge}(u, i) \leftarrow \texttt{charge}(u, i) + 1$;

**6**      $\texttt{charge}(\texttt{closest}(u, i), i + 3)) \leftarrow \texttt{charge}(\texttt{closest}(u, i), i + 3) + 1$;

**7**      **if** <u>$\texttt{first}(u, i) \neq \texttt{closest}(u, i) \wedge$ i-relevant$(\texttt{first}(u, i), u, i - 1)$</u> **then**

**8**          Add edge $u, \texttt{first}(u, i)$ to $E$;

**9**          $\texttt{charge}(u, i - 1) \leftarrow \texttt{charge}(u, i - 1) + 1$;

**10**      **end**

**11**      **if** <u>$\texttt{last}(u, i) \neq \texttt{closest}(u, i) \wedge$ i-relevant$(\texttt{last}(u, i), u, i + 1)$</u> **then**

**12**          Add edge $\{u, \texttt{last}(u, i)\}$ to $E$;

**13**          $\texttt{charge}(u, i + 1) \leftarrow \texttt{charge}(u, i + 1) + 1$;

**14**      **end**

**15** **end**

**16** **foreach** <u>cone $i$ of vertex $u \in DT$ where $i \in \{0, 2, 4\} \wedge$ i-distant$(u, i)$ is true</u> **do**

**17**      $v_{next} \leftarrow \texttt{first}(u, i + 1)$;

**18**      $v_{prev} \leftarrow \texttt{last}(u, i - 1)$;

**19**      Add edge $\{v_{next}, v_{prev}\}$ to $E$;

**20**      $\texttt{charge}(v_{next}, i + 1) \leftarrow \texttt{charge}(v_{next}, i + 1) + 1$;

**21**      $\texttt{charge}(v_{prev}, i - 1) \leftarrow \texttt{charge}(v_{prev}, i - 1) + 1$;

**22**      Let $v_{remove}$ be the vertex from $v_{next}, v_{prev}$ where $\angle(\texttt{parent}(u, i), u, v_{remove})$ is maximized;

**23**      Remove edge $\{u, v_{remove}\}$ from $E$;

**24**      $\texttt{charge}(u, i) \leftarrow \texttt{charge}(u, i) - 1$;

**25** **end**

**26** **foreach** <u>cone $i$ of vertex $u \in DT$ where $i \in \{0, 1, \ldots, 5\}$</u> **do**

**27**      **if** <u>$\texttt{charge}(u, i)$ is not $2 \vee \texttt{charge}(u, i - 1)$ is not $1 \vee \texttt{charge}(u, i + 1)$ is not $1$</u> **then**

**28**          break;

**29**      **end**

**30**      **if** <u>$u = \texttt{last}(\texttt{parent}(u, i), i)$</u> **then**

**31**          $v_{remove} \leftarrow \texttt{last}(u, i - 1)$;

**32**      **else**

**33**          $v_{remove} \leftarrow \texttt{first}(u, i + 1)$;

**34**      **end**

**35**      Remove edge $\{u, v_{remove}\}$ from $E$;

**36**      $\texttt{charge}(u, i) \leftarrow \texttt{charge}(u, i) - 1$;

**37** **end**

**38** **return** $E$;

---

## Algorithm 17: i-relevant($v, u, i$)

**1** $w \leftarrow \texttt{parent}(u, i)$;

**2** **return** <u>$v \neq \texttt{closest}(u, i) \wedge v \in C_w^i$</u>;

**Algorithm 18:** `i-distant`$(w, i)$

---

**1** $u \leftarrow \text{parent}(w, i);$

**2** **return**
$\{w, u\} \notin E \land \text{i-relevant}(\text{first}(w, i+1), u, i+1) \land \text{i-relevant}(\text{last}(w, i-1), u, i-1);$

---

**Algorithm 19:** `parent`$(u, i)$

---

**1** **return** the closest vertex to $u$ in a even cone $i$, if it exists;

---

**Algorithm 20:** `closest`$(u, i)$

---

**1** **return** the closest vertex to $u$ in a odd cone $i$, if it exists;

---

**Algorithm 21:** `first`$(u, i)$

---

**1** **return** the first vertex in a odd cone $i$, if it exists;

---

**Algorithm 22:** `last`$(u, i)$

---

**1** **return** the last vertex in a odd cone $i$, if it exists;

---

**Algorithm 23:** `charge`$(u, i)$

---

**1** **return** the number of edges charged to cone $i$ of $u$;

---

**Algorithm 24:** `BKPX15`$(P)$

---

**1** $DT \leftarrow L_\infty\text{-DelaunayTriangulation}(P);$

**2** $\overrightarrow{Y_4^\infty} \leftarrow \text{constructYaoInfinityGraph}(DT);$

**3** $A \leftarrow \text{selectAnchors}(\overrightarrow{Y_4^\infty}, DT);$

**4** $H_8 \leftarrow \text{degree8Spanner}(A, \overrightarrow{Y_4^\infty}, DT);$

**5** $H_6 \leftarrow \text{processDupEdgeChains}(H_8, \overrightarrow{Y_4^\infty});$

**6** $H_4 \leftarrow \text{createShortcuts}(H_6, \overrightarrow{Y_4^\infty}, DT);$

**7** **return** $H_4;$

---

**Algorithm 25:** `constructYaoInfinityGraph`$(DT)$

---

**1** $\overrightarrow{Y_4^\infty} \leftarrow \emptyset;$

**2** **foreach** $u \in DT$ **do**

**3**    **foreach** $C_u^i$ **do**

**4**       Let $v \in C_u^i$ be the vertex with the smallest $L_\infty$ distance;

**5**       Add $(u, v)$ to $\overrightarrow{Y_4^\infty};$

**6**    **end**

**7** **end**

**8** **return** $\overrightarrow{Y_4^\infty};$

## Algorithm 26: `selectAnchors`$(\overrightarrow{Y_4^\infty}, DT)$

**1**  **foreach** $(u, v) \in \overrightarrow{Y_4^\infty}$ **do**
**2**      Let $i$ be the cone of $u$ containing $v$;
**3**      $v_{anchor} \leftarrow v$;
**4**      **if** $\neg \texttt{isMutuallySingle}(\overrightarrow{Y_4^\infty}, u, v, i)$ **and** $u$ has more than one $\overrightarrow{Y_4^\infty}$ edge in $C_u^i$ **then**
**5**          Let $\ell$ be the position of $v$ and $k$ the number of vertices in $\texttt{fan}(DT, u, i)$;
**6**          **if** $\ell \geq 2$ **and** $(v_{\ell-1}, v_\ell) \in \overrightarrow{Y_4^\infty}$ **and** $(v_\ell, v_{\ell-1}) \notin \overrightarrow{Y_4^\infty}$ **then**
**7**              Let $v_{\ell'}$ such that $\ell' < \ell$ be the starting vertex of the maximal unidirectional `canonical` path ending at $v_\ell$;
**8**              $v_{anchor} \leftarrow v_{\ell'}$;
**9**          **else if** $\ell \leq k - 1$ **and** $(v_{\ell+1}, v_\ell) \in \overrightarrow{Y_4^\infty}$ **and** $(v_\ell, v_{\ell+1}) \notin \overrightarrow{Y_4^\infty}$ **then**
**10**              Let $v_{\ell'}$ such that $\ell' > \ell$ be the starting vertex of the maximal unidirectional `canonical` path ending at $v_\ell$;
**11**              $v_{anchor} \leftarrow v_{\ell'}$;
**12**      **end**
**13**      Mark $(u, v_{anchor})$ as the `anchor` of $C_u^i$;
**14**  **end**
**15**  $A \leftarrow \emptyset$;
**16**  **foreach** anchor $(u, v)$ in each $C_u^i$ **do**
**17**      **if** anchor of $C_v^{i+2}$ is $(v, u)$ or undefined **then**
**18**          Mark $(u, v)$ as `strong` and add it to $A$;
**19**      **else**
**20**          Mark $(u, v)$ as `weak`;
**21**      **end**
**22**  **end**
**23**  **foreach** weak anchor $(u, v)$ in each $C_u^i$ **do**
**24**      **if** $u$ begins the weak anchor chain $(w_0, w_1, ..., w_k)$ **then**
**25**          **if** $k$ is odd **then**
**26**              Mark $(w_0, w_1)$ as a `start-of-odd-chain-anchor`;
**27**          **end**
**28**          **for** $\ell = k - 1;\ \ell \geq 0;\ \ell = \ell - 2$ **do**
**29**              Add $(w_{\ell-1}, w_\ell)$ to $A$;
**30**          **end**
**31**      **end**
**32**  **end**
**33**  **return** $A$;

## Algorithm 27: `fan`$(DT, u, i)$

**1**  **return** all neighboring vertices $(v_1, v_2, ..., v_k)$ in $C_u^i$ in counterclockwise order;

## Algorithm 28: `isMutuallySingle`$(\overrightarrow{Y_4^\infty}, u, v, i)$

**1**  **return** $u$ has one $\overrightarrow{Y_4^\infty}$ edge in $C_u^i$ **and** $v$ has one $\overrightarrow{Y_4^\infty}$ edge in $C_v^{i+2}$;

---

**Algorithm 29:** degree8Spanner($A, \overrightarrow{Y_4^\infty}, DT$)

---

**1** Charge each anchor $(u, v) \in A$ to the cones of each vertex in which the edge lies;

**2** $H_8 \leftarrow A$;

**3** **foreach** <u>vertex $u$ and cone $i$ of $u$</u> **do**

**4**     $\{v_1, ..., v_k\} \leftarrow$ fan$(DT, u, i)$;

**5**     **if** <u>$k \geq 2$</u> **then**

**6**        Add all uni-directional canonical edges to $H_8$ except $(v_2, v_1)$ and $(v_{k-1}, v_k)$;

**7**        Add $(v_2, v_1)$ to $H_8$ **if** $(v_2, v_1)$ is a non-anchor, uni-directional edge such that
         $(v_2, v_1) \in \overrightarrow{Y_4^\infty} \wedge (v_1, v_2) \notin \overrightarrow{Y_4^\infty} \wedge (v_1, u)$ is a dual edge $\wedge (v_1, u)$ is not a
         start-of-odd-chain anchor chosen by $v_1$;

**8**        Add $(v_{k-1}, v_k)$ to $H_8$ **if** $(v_{k-1}, v_k)$ is a non-anchor, uni-directional edge such that
         $(v_{k-1}, v_k) \in \overrightarrow{Y_4^\infty} \wedge (v_k, v_{k-1}) \notin \overrightarrow{Y_4^\infty} \wedge (v_k, u)$ is a dual edge $\wedge (v_k, u)$ is not a
         start-of-odd-chain anchor chosen by $v_k$;

**9**        **foreach** <u>canonical edge $(v, w)$ added to $H_8$</u> **do**

**10**           $v_{charge} \leftarrow v$;

**11**           **if** <u>$(v, w)$ is a non-anchor</u> **then**

**12**             $v_{charge} \leftarrow u$;

**13**           **end**

**14**           Charge $(v, w)$ to the cone of $v$ containing $w$ and the cone of $w$ containing $v_{charge}$;

**15**        **end**

**16**     **end**

**17** **end**

**18** **return** $H_8$;

---

 

---

**Algorithm 30:** createShortcuts($H_6, \overrightarrow{Y_4^\infty}, DT$)

---

**1** $H_4 \leftarrow H_6$;

**2** **foreach**
    <u>pair of non-anchor uni-directional canonical edges $(v_{r-1}, v_r), (v_{r+1}, v_r)$ in cone $i$ of $u$</u> **do**

**3**     Remove $(v_{r-1}, v_r)$ and $(v_{r+1}, v_r)$ from $H_4$;

**4**     Add $(v_{r-1}, v_{r+1})$ to $H_4$;

**5**     Charge this edge to the cones of each vertex in which the edge lies;

**6** **end**

**7** **return** $H_4$;

---

**Algorithm 31:** processDupEdgeChains($H_8, \overrightarrow{Y_4^\infty}$)

1   $H_6 \leftarrow H_8$;

2   **foreach** <u>uni-directional non-anchor $(u, v)$ in cone $i$ of $u$ in $H_8$ with charge $= 1$</u> **do**

3     **if** <u>cone $i + 1$ or $i - 1$ of $v$ has charge $= 2 \wedge (u, v)$ is charged to cone $i + 1$ or $i - 1$ of $v$</u>
     **then**

4       Let $j$ be the cone of $v$ where $(u, v)$ is charged;

5       $v_{current} \leftarrow u, v_{next} \leftarrow v$;

6       $D \leftarrow \emptyset$;

7       **while** <u>cone $j$ of $v_{next}$ has charge $= 2 \wedge (v_{current}, v_{next})$ is in cone $j$ of $v_{next}$</u> **do**

8        Add $(v_{current}, v_{next})$ to $D$;

9        $v_{current} \leftarrow v_{next}$;

10        Set $v_{next}$ to the target of the $\overrightarrow{Y_4^\infty}$ edge beginning in cone $j$ of $v_{current}$;

11        swap($i, j$);

12       **end**

13       Starting with the last edge in the path induced by $D$, remove every other edge from
       $H_6$;

14     **end**

15   **end**

16   **return** $H_6$;

**Algorithm 32:** KPT17($P$)

---

**1** $DT \leftarrow$ TD-DelaunayTriangulation($P$);

**2** $E, A \leftarrow \emptyset$;

**3** **foreach** white anchor $(u, v)$ in increasing order of $d_\triangledown$ length **do**

**4**     **if** $u$ and $v$ do not have a white anchor in a cone adjacent to $(u, v)$'s cone **then**

**5**         | Add $(u, v)$ to $A$;

**6**     **end**

**7** **end**

**8** Add all blue anchors to $A$;

**9** **foreach** blue anchor $u$ **do**

**10**     Let $s_1, s_2, ..., s_m$ be the clockwise ordered neighbors of $u$ in $DT$;

**11**     Add all canonical edges $(s_\ell, s_{\ell+1}) \notin A$ to $E$ such that $1 \le \ell < m$;

**12** **end**

**13** **foreach** pair of canonical edges $(u, v), (w, v) \in E$ in a blue cone **do**

**14**     Remove $(u, v)$ and $(w, v)$ from $E$;

**15**     Add a shortcut edge $(u, w)$ to E;

**16** **end**

**17** **foreach** white canonical edge $(u, v)$ on the white side of its anchor $a$ **do**

**18**     **if** $a \notin A$ **then**

**19**         | Add $(u, v)$ to $E$;

**20**     **end**

**21** **end**

**22** **foreach** white anchor $(v, w)$ and its boundary edge $(u, w) \ne (v, w)$ on the white side **do**

**23**     Let $u = s_1, s_2, ..., s_m = v$ be the canonical path between $u$ and $v$;

**24**     **for** $i = 0$ **to** $m$ **do**

**25**         **if** $(s_{i+1}, s_i)$ is blue **then**

**26**             Let $j$ be the smallest index in $P_i = \{s_{i+1}, ..., s_m\}$ such that $s_j$ is in a white cone of $s_i$ and $P_i$ lies on the same side (or on) the straight line $s_i s_j$;

**27**             Add the shortcut $(s_j, s_i)$ to $E$;

**28**             **if** $(s_j, s_{j-1}) \in E$ **then**

**29**                | Remove $(s_j, s_{j-1})$ from $E$;

**30**             **end**

**31**             $i \leftarrow j$;

**32**         **end**

**33**     **end**

**34** **end**

**35** **return** $E \cup A$;

**Algorithm 33:** $\texttt{Wedge}_6(u, v_i)$

---

**1** **foreach** $C_u^z$ containing $\{u, v_i\}$ **do**

**2** $\quad$ Let $Q = \{v_n : \{u, v_n\} \in C_u^z \cap DT\} = \{v_j, ..., v_k\}$;

**3** $\quad$ Let $Q' = \{v_n : \angle v_{n-1} v_n v_{n+1} < 6\pi/7, v_n \in Q \setminus \{v_j, v_i, v_k\}\}$;

**4** $\quad$ Add all edges $\{v_n, v_{n+1}\}$ to $E*$ such that $v_n, v_{n+1} \notin Q'$ and
$\quad\quad n \in [j+1, i-2] \cup [i+1, k-2]$;
$\quad$ /* W.l.o.g. the points of $Q'$ lie between $v_i$ and $v_k$ (the symmetric case is
$\quad\quad$ handled analogously) $\hfill$ */

**5** $\quad$ **if** $\angle u v_i v_{i-1} > 4\pi/7$ and $i, i-1 \neq j$ **then**

**6** $\quad\quad$ | $\quad$ Add edge $\{v_i, v_{i-1}\}$ to $E*$;

**7** $\quad$ **end**

**8** $\quad$ Let $v_f$ be the first point in $Q'$;

**9** $\quad$ Let $a = \min\{n | n > f$ and $v_n \in Q \setminus Q'\}$;

**10** $\quad$ **if** $f = i+1$ **then**

**11** $\quad\quad$ **if** $\angle u v_i v_{i+1} \leq 4\pi/7$ and $a \neq k$ **then**

**12** $\quad\quad\quad$ | $\quad$ Add edge $\{v_f, v_a\}$ to $E*$;

**13** $\quad\quad$ **end**

**14** $\quad\quad$ **if** $\angle u v_i v_{i+1} > 4\pi/7$ and $f+1 \neq k$ **then**

**15** $\quad\quad\quad$ | $\quad$ Add edge $\{v_i, v_{f+1}\}$ to $E*$;

**16** $\quad\quad$ **end**

**17** $\quad$ **else**

**18** $\quad\quad$ Let $v_\ell$ be the last point in $Q'$;

**19** $\quad\quad$ Let $b = \max\{n | n < \ell$ and $v_n \in Q \setminus Q'\}$;

**20** $\quad\quad$ **if** $\ell = k-1$ **then**

**21** $\quad\quad\quad$ | $\quad$ Add edge $\{v_\ell, v_b\}$ to $E*$;

**22** $\quad\quad$ **else**

**23** $\quad\quad\quad$ Add edge $\{v_b, v_{\ell+1}\}$ to $E*$;

**24** $\quad\quad\quad$ **if** $v_{\ell-1} \in Q'$ **then**

**25** $\quad\quad\quad\quad$ | $\quad$ Add edge $\{v_\ell, v_{\ell-1}\}$ to $E*$;

**26** $\quad\quad\quad$ **end**

**27** $\quad\quad$ **end**

**28** $\quad$ **end**

**29** **end**

ESTIMATING STRETCH FACTORS OF LARGE SPANNERS

Measuring exact stretch factors of large graphs is a tedious job, and so is for geometric spanners. Although many algorithms exist in the literature for constructing geometric spanners, nothing is known about practical algorithms for computing stretch factors of large geometric spanners. It is a severe bottleneck for conducting experiments with large spanners since the stretch factor is considered a fundamental quality of geometric spanners.

For any spanner (not necessarily geometric) on $n$ vertices, its exact stretch factor can be computed in $O(n^3 \log n)$ time by running the folklore Dijkstra algorithm (implemented using a Fibonacci heap) from every vertex, and in $\Theta(n^3)$ time by running the classic Floyd-Warshall algorithm. Note that the Dijkstra-based algorithm runs $O(n^2 \log n)$ time for plane spanners since the number of edges is $O(n)$. Both of these are very slow in practice. However, the latter has a quadratic space-complexity and is unusable when $n$ is large. Consequently, they are practically useless when $n$ is large. Stretch factor estimation of large geometric graphs appears to be a far cry despite theoretical studies on this problem; refer to [2, 20, 28, 38, 43]. We believe these algorithms are either involved from an algorithm engineering standpoint or rely on well-separated pair decomposition [18], which may potentially slow down practical implementations due to the large number of well-separated pairs needed by those algorithms. This has motivated us to design a practical algorithm, named APPXSTRETCHFACTOR, that gives a lower-bound on the actual stretch factor of any geometric spanner (not necessarily plane). However, we will consider the universe of plane geometric spanners as the input domain in this work. To our knowledge, we are not aware of any such algorithm in the literature. Refer to Algorithm 34. It takes as input an $n$-element pointset $P$ and a geometric

graph $G$, constructed on $P$.

---

**Algorithm 34:** AppxStretchFactor$(P, G)$

---

**1** $DT \leftarrow L_2\text{-}\texttt{DelaunayTriangulation}(P)$;

**2** $t \leftarrow 1$;

**3 foreach** $p \in P$ **do**

**4**      $h \leftarrow 1, t_p \leftarrow 1$;

**5**      **while** true **do**

**6**          Let $X$ denote the set of points which are exactly $h$ hops away from $p$ in $DT$ found using a breadth-first traversal originating at $u$;

**7**          $t' \leftarrow 1$;

**8**          **foreach** $q \in X$ **do**

**9**              $t' \leftarrow \max\left(\frac{|\pi_G(p,q)|}{|pq|}, t'\right)$;

**10**          **end**

**11**          **if** $t' > t_p$ **then**

**12**              $h \leftarrow h + 1; t_p \leftarrow t'$;

**13**          **else**

**14**              **break**;

**15**      **end**

**16**      $t \leftarrow \max(t, t_p)$;

**17 end**

**18 return** $t$;

---

The underlying idea of our algorithm is as follows. We observe that most geometric spanners are well-constructed; meaning it is likely that far away points (having many hops in the shortest paths between them) have low detour ratios (ratio of the length of a shortest path to that of the Euclidean distance) between them and the worst-case detour is achieved by point pairs that are a few hops apart. Note that stretch factor of a graph is the maximum detour ratio over all vertex pairs. To capture *closeness*, we use the $L_2$-Delaunay triangulation constructed on $P$ as the basis. For every point $p \in P$, we start a breadth-first traversal on the Delaunay triangulation $DT$. At every level, we compute the detour ratios in $G$ from $p$ to all the points in that level. If a worse detour ratio is found in the current level compared to the worst found in the previous level, we continue to the next level; otherwise, the process is terminated. For finding detour ratios in $G$, we use the folklore Dijkstra algorithm since

computation of shortest paths are required. In our algorithm, $\pi_G(p, q)$ denotes a shortest path between the points $p, q \in P$ in $G$ and $|\pi_G(p, q)|$ its total length. The detour between $p, q$ in $G$ can be easily calculated as $|\pi_G(p, q)|/|pq|$. The current level is denoted by $h$. It is assumed that the neighbors of $p$ in $G$ are at level 1. For efficiency reasons, we do not restart the Dijkstra at every level of the breadth-first traversal; instead, we save our progress from the previous level and continue after that.

To our surprise, we find that for the class of spanners used in this work, APPXSTRETCH-FACTOR returned exact stretch factors almost every time. The precision error was very low whenever it failed to compute the exact stretch factor. Further, our algorithm can be parallelized very easily by spawning parallel iterations of the **foreach** loop. Apart from the $L_2$-Delaunay triangulation (which can be constructed very fast in practice), it does not use any advanced geometric structure, making it fast in practice. We present our experimental observations for this algorithm in Section 4.3.

CHAPTER 4

EXPERIMENTS

We have implemented the algorithms in GNU C++17 using the CGAL library [41]. The machine used for experiments is equipped with a AMD Ryzen 5 1600 (3.2 GHz) processor and 24 GB of main memory, and runs Ubuntu Linux 20.04 LTS. The g++ compiler was invoked with -O3 flag to achieve fast real-world speed. The `Exact_predicates_inexact_constructions_kernel` from CGAL is used for accuracy and speed. We have tried our best to tune our codes to run fast.

All the eleven algorithms considered in this work use one of the following three kinds of Delaunay triangulation as the starting point: $L_2$, $TD$, and $L_\infty$. For constructing $L_2$ and $L_\infty$-Delaunay triangulations, we have used the `CGAL::Delaunay_triangulation_2` and `CGAL::Segment_Delaunay_graph_Linf_2` implementations, respectively. As of now, a $TD$-Delaunay triangulation implementation is not available in the CGAL. It was pointed out by Chew in [22] that such triangulations can be constructed in $O(n \log n)$ time. However, no precise implementable algorithm was presented. But luckily, it is shown in [8] by Bonichon et al. that $TD$-Delaunay triangulation of a pointset is same as its $\frac{1}{2}$-$\Theta$ graph. We leveraged this result and used the $O(n \log n)$ time `CGAL::Construct_theta_graph_2` implementation for constructing the $TD$-Delaunay triangulations. For faster speed, the input pointsets are always sorted using `CGAL::spatial_sort` before constructing Delaunay triangulations on them. We share the implementations via GitHub[1]. In our experiments, we have used both synthetic and real-world pointsets.
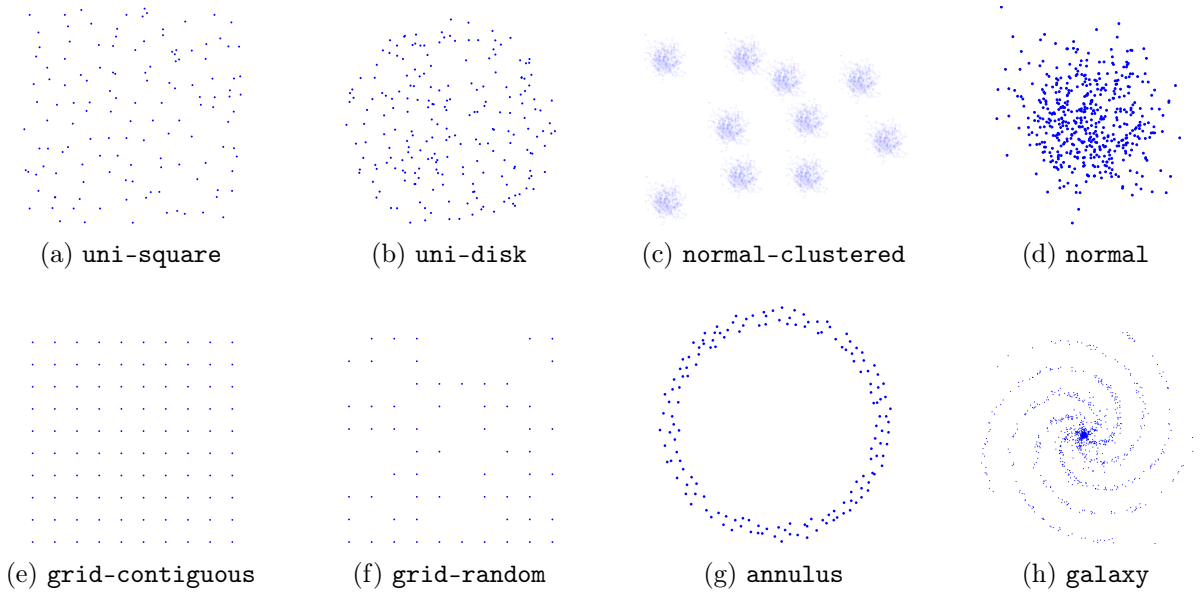
---

[1]https://github.com/mgatc/geometric-spanners

Figure 4.1: The eight distributions used to generate pointsets for our experiments.

## 4.1 Synthetic pointsets

We have used the following eight distributions to generate synthetic pointsets for our experiments. The selection of these distributions are inspired by the ones used in [5, 30, 40] for geometric experiments. See Figure 4.1 to visualize these eight distributions.

(a) `uni-square`. Points are generated uniformly inside a square of side length of 1000 using the `CGAL::Random_points_in_square_2` generator.

(b) `uni-disk`. Points are generated uniformly inside a disc of radius 1000 using the `CGAL::Random_points_in_disc_2` generator.

(c) `normal-clustered`. A set of 10 normally distributed clusters placed randomly in the plane. Each cluster contains $n/10$ normally distributed points (mean and standard-deviation are set to 2.0). We have used `std::normal_distribution<double>` to generate the point coordinates.

(d) `normal`. This is same as `normal-clustered` except that only one cluster is used.

(e) `grid-contiguous`. Points are generated contiguously on a $\lceil \sqrt{n} \rceil \times \lceil \sqrt{n} \rceil$ square grid using the `CGAL::points_on_square_grid_2` generator.

(f) `grid-random`. Points are generated on a $\lceil 0.7n \rceil \times \lceil 0.7n \rceil$ unit square grid. The value 0.7 is chosen arbitrarily to obtain well-separated non-contiguous grid points. The coordinates of the generated points are integers and are generated independently using `std::uniform_int_distribution`.

(g) `annulus`. Points are generated inside an annulus whose outer radius is set to 1000 and the inner-radius is set to 800 using `std::uniform_real_distribution`.

(h) `galaxy`. Points are generated in the shape of a spiral galaxy using `std::rand()` for randomness; see [29].

For seeding the random number generators from `C++`, we have used the Mersenne twister engine `std::mt19937`. Since some of the algorithms assume that no two points must have the same value $x$ or $y$-coordinates, the generated pointsets have been perturbed using the `CGAL::perturb_points_2` function with $0.0001, 0.0001$ as the two required parameters.

## 4.2 Real-world pointsets

The following real-world pointsets are obtained from various publicly available sources. We have removed duplicate points (wherever present) from the pointsets. The main reason behind the use of such pointsets is that they do not follow the popular synthetic distributions. Hence, experimenting with them is beneficial to see how the algorithms perform on them.

- `burma` [1]. An $33,708$-element pointset representing cities in Burma.

- `birch3` [17,31]. An $99,801$-element pointset representing random clusters at random locations.

- `monalisa` [1,31]: A $100,000$-city TSP instance representing a continuous-line drawing of the Mona Lisa.

- `KDDCU2D` [17, 31]. An 104, 297-element pointset representing the first two dimensions of a protein data-set.

- `usa` [1, 31]. A 115, 475-city TSP instance representing (nearly) all towns, villages, and cities in the United States.

- `europe` [17, 31]. An 168, 896-element pointset representing differential coordinates of the map of Europe.

- `wiki`[2]. An 317, 695-element pointset of coordinates found in English-language Wikipedia articles.

- `vlsi` [1]. An 744, 710-element pointset representing a Very Large Scale Integration chip.

- `china` [17, 31]. An 808, 693-element pointset representing cities in China.

- `uber`[3]. An 1, 381, 253-element pointset representing Uber pickup locations in New York City.

- `world` [1, 31]. An 1, 904, 711-element pointset representing all locations in the world that are registered as populated cities or towns, as well as several research bases in Antarctica.

## 4.3   Efficacy of APPXSTRETCHFACTOR

We have seen in Section 3, it is quite challenging to measure stretch factor of large spanners. This motivated us to design and use the APPXSTRETCHFACTOR algorithm in our experiments for estimating stretch factors of the generated spanners. In the following, we compare APPXSTRETCHFACTOR with the Dijkstra's algorithm (run from every vertex) and show that

---

[2] `https://github.com/placemarkt/wiki_coordinates`
[3] `https://www.kaggle.com/fivethirtyeight/uber-pickups-in-new-york-city`

for the eight distributions it is not only much faster than Dijkstra but can also estimate stretch factors of plane spanners with high accuracy.

The main reason behind the fast practical performance of AppxStretchFactor is early terminations of the breadth-first traversals (one traversal per vertex), which in turn makes Dijkstra run fast to find the shortest paths to the vertices in all the levels. We have noticed in our experiments that the pair that achieves the stretch factor for a bounded-degree plane spanner are typically a few hops away and pairwise stretch factors (ratio of detour between two vertices to that of their Euclidean distance) drop with the increase in hops. Consequently, the breadth-first traversals terminate very early most of the time.

The total number of pointsets used in this comparison experiment is $11 \cdot 8 \cdot 10 \cdot 5 = 4400$ since there are 11 algorithms, 8 distributions, 10 distinct values of $n$ $(1K, 2K, ..., 10K)$, and 5 samples were used for every value of $n$. Out of these many, the number of times AppxStretchFactor has failed to return the exact stretch factor is just 8. So, the observed failure rate is $\approx 0.18\%$. Interestingly, in the cases where AppxStretchFactor failed to compute the exact stretch factor, the largest observed error percentage between the exact stretch factor (found using Dijkstra) and the stretch factor returned by it is just $\approx 0.15$. This gives us the confidence that our algorithm can be safely used to estimate stretch factor of large spanners. Refer to the Figure 4.3. As evident from these graphs, AppxStretchFactor is substantially faster than Dijkstra everywhere. Henceforth, we use AppxStretchFactor (Algorithm 34) to estimate the stretch factors of the spanners in our experiments.
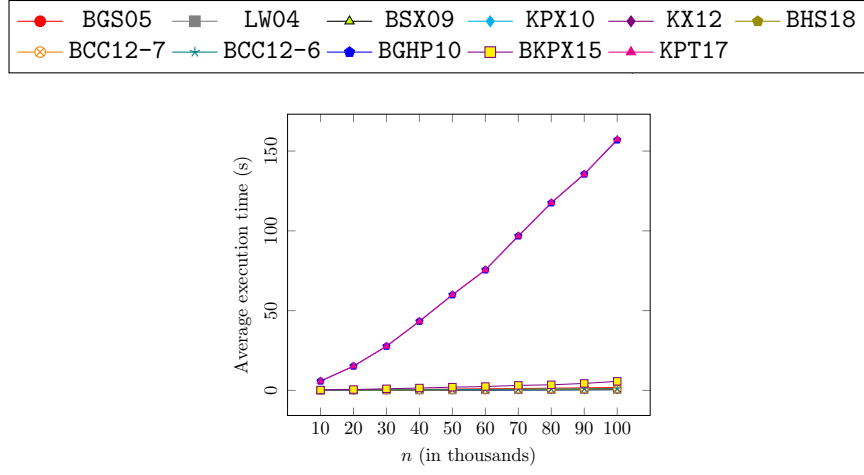
Figure 4.2: Points are generated using the `uni-square` distribution. For this particular experiment, $n$ is in the range $10K, 20K, \ldots, 100K$. The plots for `BGHP10` and `KPT17` have overlapped in this figure.

## 4.4   Experimental comparison of the algorithms

We compare the 11 implemented algorithms based on their runtimes and degree, stretch factor, and lightness[4] of the generated spanners.

For synthetic pointsets, we vary $n$ from $10K$ to $100K$. For every value of $n$, we have used five random samples to measure runtimes and the above characteristics of the spanners. In the case of real-world pointsets, we run every one of them five times and report the average time taken.

In our experiments, we find that `BGHP10` and `KPT17` are considerably slower than the other algorithms considered in this work. The reason behind this is slow construction of $TD$-Delaunay triangulations. Refer to Fig. 4.2 for an illustration. When $n = 100K$, both take more than 150 seconds to finish. In contrast, other nine algorithms take less than 10 seconds. Since real-world speed is an important factor for spanner construction algorithms, we do not

---

[4]The lightness of a geometric graph $G$ on a pointset $P$ is defined as ratio of the weight of $G$ to that of a Euclidean minimum spanning tree on $P$.

consider them further in our runtime comparisons.

1. *Runtime.* Fast execution speed is highly desired for spanner construction on large pointsets. We present the runtimes for all eight distributions in Fig. 4.4. As explained above, we have excluded `BGHP10` and `KPT17` from these plots since they are considerably slower than the other nine algorithms. Interestingly, we find that the relative performance of these algorithms is independent of the point distributions. For all the eight distributions, we find that `BKPX15` is much slower than the others. This is mainly due to the time taken to construct $L_\infty$-Delauanay triangulations. Among the ones that use $L_2$-Delauanay triangulations, `BGS05` is the slowest due to the overhead of creation of temporary geometric graphs needed to control the degree and stretch-factor of the ouput spanners. Refer to Section 2 to see more details on this algorithm. The fastest algorithms are `KPX10`, `BSX09`, `LW04`, and `KX12`. The main reason behind their speedy performance is fast construction of $L_2$-Delaunay triangulations and lightweight processing of the triangulations for spanner construction. The `BHS18`, `BCC12-7`, and `BCC12-6` algorithms came out quite close the above four algorithms. Note that these three algorithms also use $L_2$-Delaunay triangulation as the starting point. The same observations hold for the real-world pointsets used in our experiments. See Table 4.3 for the runtimes in seconds.

2. *Degree.* Refer to Table 4.1. In the tables, $\Delta$ denotes the theoretical degree upper bound, as claimed by the authors of these algorithms, max $\Delta_{\text{observed}}$ denotes the maximum degree observed in our experiments, avg $\Delta_{\text{observed}}$ denotes the observed average degree, and avg $\Delta_{\text{vertex}}$ denotes the observed average degree per vertex. In our experiments, we find that spanners generated by `BGS05`, `LW04`, and `BSX09` have degrees much less than the degree upper-bounds derived by the authors. While it cannot be denied that there could be special examples where these upper-bounds are actually achieved, the maximum degrees achieved in our experiments are 14, 11, and 9, respectively. Note that the theoretical degree upper bounds are 27, 23, and 17, respectively. For the remaining

eight algorithms, the claimed degree upper-bounds are achieved in our experiments thereby showing the analyses obtained by the authors of those algorithms are tight. However, the degree bound claimed by the authors of `BCC12-6` appears incorrect. We present an example in Appendix B where the degree of the spanner generated by this algorithm exceeds 6 (in fact, it is 7 in this example). For every algorithm, we find that the average degree of the generated spanners is not far away from the maximum observed degrees. It shows that the algorithms are consistent in constructing the spanners. The average degree per vertex is another way to judge the quality of the spanners. In this regard, we find that it is always between 6 and 3 everywhere and is quite reasonable for practical purposes. This shows that all these algorithms are very careful when it comes to the selection of spanner edges. The lowest values are achieved by `BKPX15` and `KPT17`. For the real-world pointsets, we find similar performance from the algorithms when it comes to the degree and degree per vertex of the spanners. This is quite surprising since these real-world pointsets do not follow specific distributions. Refer to Tables 4.4 and 4.5 for more details. Note that `BSG05` has produced a degree-15 spanner for the `vlsi` pointset. In contrast, for the synthetic pointsets the highest degree we could observe is 14.

3. *Stretch factor.* Refer to Table 4.2. In the tables, $t$ denotes the theoretical stretch factor, as derived by the authors of these algorithms; $t_{\max}$ denotes the maximum stretch factor observed in our experiments, and $t_{\mathrm{avg}}$ denotes the average observed stretch factor. Among the eleven algorithms, `KPX10` has the lowest guaranteed stretch factor - it is 2.9. The stretch factors of the spanners generated by `KPX10` are always less than 1.6, thereby making it the best among the eleven algorithms in terms of stretch factor. In this regard, `BKPX15` turned out to be the worst; the largest stretch factor we have observed is 7.242, although it is substantially less than the theoretical stretch factor upper bound of 156.8. Its competitor `KPX17` that can also generate degree-4 plane spanners has a lower observed maximum stretch factor - it is 5.236 (theoretical upper bound is 20 for

this algorithm). Overall, we find that the stretch factors of the generated spanners are much less than the claimed theoretical upper bounds. This shows that the generated spanners are well-constructed in practice. With the exception of BKPX15, we find that the average stretch factors are quite close to the maximum stretch factors. Now let us turn our attention to the real-world pointsets. Refer to Table 4.6. Once again KPX10 produced lowest stretch factor spanners. The stretch factors seem quite reasonable everywhere except the two cases of vlsi and uber pointsets when fed to BKPX15. The produced spanners have stretch factors of 11.535 and 27.929, respectively. The later is interesting since the lower bound example constructed by the authors in [11] for the worst-case stretch factor of the spanners produced by BKPX15 has a stretch factor of $7 + 7\sqrt{2} \approx 16.899$. The uber pointset beats this lower bound.

4. *Lightness.* Since a minimum spanning tree is the cheapest (in terms of the sum of the total length of the edges) way to connect $n$ points, lightness can be used to judge the quality of spanners. This metric is beneficial when spanners are used for constructing computer or transportation networks. Refer to Table 4.2. Lightness is denoted by $\ell$. With a few exceptions, we find that lightness somewhat correlates with degree. This is because using a lower number of carefully placed spanner edges usually leads to lower lightness. The spanners generated by BGS05 are always found to have the highest lightness. This is expected because of their high degrees. Although, the difference in degree of the spanners generate by BGS05 and LW04 is marginal (around 2), the difference between their lightness is substantial (approximately 6 for some cases). On the other hand, the degree-4 spanners generated by KPT17 have the lowest lightnesses (less than 2.9 everywhere). Interestingly, although BKPX15 generates degree-4 spanners, their lightness is found to be approximately twice that of the ones generated by KPT17. In fact, their lightness turned out to be one of the highest. This shows that KPT17 is more careful when it comes to placing long edges. The lightnesses of the spanners generated for real-world pointsets follow a similar trend, and we did not observe anything special.

Refer to Table 4.7 for more details.

**Remark.** In our experiments, we find that the spanners' degree, stretch factor, and lightness remained somewhat constant with the increase in $n$. Hence, we do not present plots for them.

| Algorithm | max $\Delta_{\text{observed}}$ | avg $\Delta_{\text{observed}}$ | avg $\Delta_{\text{vertex}}$ |
|---|---|---|---|
| BGS05 | 13 | 11.580 | 5.745 |
| LW04 | 11 | 9.820 | 5.148 |
| BSX09 | 9 | 7.840 | 4.392 |
| KPX10 | 14 | 12.980 | 5.994 |
| KX12 | 10 | 9.960 | 5.449 |
| BHS18 | 7 | 6.980 | 4.113 |
| BCC12-7 | 7 | 7.000 | 4.337 |
| BCC12-6 | 7 | 6.820 | 4.016 |
| BGHP10 | 6 | 6.000 | 4.240 |
| BKPX15 | 4 | 4.000 | 3.328 |
| KPT17 | 4 | 4.000 | 3.140 |

(a) `uni-square`

| Algorithm | max $\Delta_{\text{observed}}$ | avg $\Delta_{\text{observed}}$ | avg $\Delta_{\text{vertex}}$ |
|---|---|---|---|
| BGS05 | 12 | 11.540 | 5.744 |
| LW04 | 11 | 9.880 | 5.146 |
| BSX09 | 9 | 7.760 | 4.389 |
| KPX10 | 13 | 12.420 | 5.993 |
| KX12 | 10 | 9.980 | 5.451 |
| BHS18 | 7 | 6.200 | 4.109 |
| BCC12-7 | 7 | 7.000 | 4.335 |
| BCC12-6 | 7 | 6.020 | 4.012 |
| BGHP10 | 6 | 6.000 | 4.239 |
| BKPX15 | 4 | 4.000 | 3.319 |
| KPT17 | 4 | 4.000 | 3.141 |

(b) `uni-disk`

| Algorithm | max $\Delta_{\text{observed}}$ | avg $\Delta_{\text{observed}}$ | avg $\Delta_{\text{vertex}}$ |
|---|---|---|---|
| BGS05 | 13 | 11.600 | 5.750 |
| LW04 | 11 | 9.800 | 5.154 |
| BSX09 | 9 | 7.860 | 4.392 |
| KPX10 | 14 | 12.820 | 5.996 |
| KX12 | 10 | 9.980 | 5.450 |
| BHS18 | 7 | 6.100 | 4.110 |
| BCC12-7 | 7 | 7.000 | 4.334 |
| BCC12-6 | 6 | 6.000 | 4.010 |
| BGHP10 | 6 | 6.000 | 4.245 |
| BKPX15 | 4 | 4.000 | 3.327 |
| KPT17 | 4 | 4.000 | 3.144 |

(c) `normal-clustered`

| Algorithm | max $\Delta_{\text{observed}}$ | avg $\Delta_{\text{observed}}$ | avg $\Delta_{\text{vertex}}$ |
|---|---|---|---|
| BGS05 | 13 | 11.740 | 5.751 |
| LW04 | 11 | 9.760 | 5.158 |
| BSX09 | 9 | 7.880 | 4.395 |
| KPX10 | 14 | 12.320 | 5.997 |
| KX12 | 10 | 9.940 | 5.452 |
| BHS18 | 7 | 6.060 | 4.117 |
| BCC12-7 | 7 | 7.000 | 4.346 |
| BCC12-6 | 6 | 6.000 | 4.023 |
| BGHP10 | 6 | 6.000 | 4.254 |
| BKPX15 | 4 | 4.000 | 3.327 |
| KPT17 | 4 | 4.000 | 3.150 |

(d) `normal`

| Algorithm | max $\Delta_{\text{observed}}$ | avg $\Delta_{\text{observed}}$ | avg $\Delta_{\text{vertex}}$ |
|---|---|---|---|
| BGS05 | 11 | 11.000 | 5.919 |
| LW04 | 9 | 7.920 | 5.099 |
| BSX09 | 9 | 7.660 | 4.412 |
| KPX10 | 12 | 11.880 | 5.993 |
| KX12 | 11 | 10.000 | 5.987 |
| BHS18 | 7 | 7.000 | 4.825 |
| BCC12-7 | 7 | 7.000 | 5.160 |
| BCC12-6 | 7 | 6.960 | 4.361 |
| BGHP10 | 6 | 6.000 | 4.979 |
| BKPX15 | 4 | 4.000 | 3.350 |
| KPT17 | 4 | 4.000 | 3.537 |

(e) `grid-contiguous`

| Algorithm | max $\Delta_{\text{observed}}$ | avg $\Delta_{\text{observed}}$ | avg $\Delta_{\text{vertex}}$ |
|---|---|---|---|
| BGS05 | 13 | 11.540 | 5.745 |
| LW04 | 11 | 9.900 | 5.148 |
| BSX09 | 9 | 7.980 | 4.391 |
| KPX10 | 14 | 12.940 | 5.995 |
| KX12 | 10 | 9.980 | 5.449 |
| BHS18 | 7 | 6.960 | 4.114 |
| BCC12-7 | 7 | 7.000 | 4.338 |
| BCC12-6 | 7 | 6.740 | 4.016 |
| BGHP10 | 6 | 6.000 | 4.240 |
| BKPX15 | 4 | 4.000 | 3.327 |
| KPT17 | 4 | 4.000 | 3.140 |

(f) `grid-random`

| Algorithm | max $\Delta_{\text{observed}}$ | avg $\Delta_{\text{observed}}$ | avg $\Delta_{\text{vertex}}$ |
|---|---|---|---|
| BGS05 | 13 | 11.520 | 5.730 |
| LW04 | 11 | 9.740 | 5.130 |
| BSX09 | 9 | 8.100 | 4.382 |
| KPX10 | 14 | 13.180 | 5.987 |
| KX12 | 11 | 10.300 | 5.448 |
| BHS18 | 7 | 6.960 | 4.098 |
| BCC12-7 | 7 | 7.000 | 4.313 |
| BCC12-6 | 8 | 6.940 | 3.994 |
| BGHP10 | 6 | 6.000 | 4.230 |
| BKPX15 | 4 | 4.000 | 3.315 |
| KPT17 | 4 | 4.000 | 3.130 |

(g) `annulus`

| Algorithm | max $\Delta_{\text{observed}}$ | avg $\Delta_{\text{observed}}$ | avg $\Delta_{\text{vertex}}$ |
|---|---|---|---|
| BGS05 | 14 | 12.320 | 5.736 |
| LW04 | 11 | 9.920 | 5.134 |
| BSX09 | 9 | 8.180 | 4.384 |
| KPX10 | 14 | 13.680 | 5.993 |
| KX12 | 11 | 10.000 | 5.434 |
| BHS18 | 7 | 6.580 | 4.090 |
| BCC12-7 | 7 | 7.000 | 4.290 |
| BCC12-6 | 7 | 6.120 | 3.970 |
| BGHP10 | 6 | 6.000 | 4.228 |
| BKPX15 | 4 | 4.000 | 3.326 |
| KPT17 | 4 | 4.000 | 3.131 |

(h) `galaxy`

Table 4.1: Degree comparisons of the spanners generated by the eleven algorithms.

| Algorithm | $t_{\max}$ | $t_{\text{avg}}$ | $\ell$ |
|---|---|---|---|
| BGS05 | 2.687 | 2.215 | 10.209 |
| LW04 | 2.687 | 2.349 | 4.366 |
| BSX09 | 4.284 | 3.666 | 3.585 |
| KPX10 | 1.519 | 1.453 | 5.378 |
| KX12 | 2.021 | 1.848 | 4.728 |
| BHS18 | 2.812 | 2.548 | 3.315 |
| BCC12-7 | 2.433 | 2.152 | 3.636 |
| BCC12-6 | 2.894 | 2.494 | 3.298 |
| BGHP10 | 3.204 | 2.755 | 3.429 |
| BKPX15 | 4.692 | 3.505 | 4.824 |
| KPT17 | 4.895 | 4.067 | 2.261 |

(a) `uni-square`

| Algorithm | $t_{\max}$ | $t_{\text{avg}}$ | $\ell$ |
|---|---|---|---|
| BGS05 | 2.409 | 2.202 | 10.014 |
| LW04 | 2.525 | 2.325 | 4.310 |
| BSX09 | 4.267 | 3.656 | 3.528 |
| KPX10 | 1.497 | 1.450 | 5.279 |
| KX12 | 2.034 | 1.857 | 4.661 |
| BHS18 | 3.152 | 2.587 | 3.275 |
| BCC12-7 | 2.460 | 2.167 | 3.597 |
| BCC12-6 | 2.998 | 2.460 | 3.261 |
| BGHP10 | 3.034 | 2.782 | 3.415 |
| BKPX15 | 4.255 | 3.501 | 4.702 |
| KPT17 | 5.050 | 4.052 | 2.259 |

(b) `uni-disk`

| Algorithm | $t_{\max}$ | $t_{\text{avg}}$ | $\ell$ |
|---|---|---|---|
| BGS05 | 2.555 | 2.214 | 12.802 |
| LW04 | 2.522 | 2.327 | 5.220 |
| BSX09 | 4.226 | 3.664 | 4.345 |
| KPX10 | 1.489 | 1.448 | 6.752 |
| KX12 | 2.126 | 1.856 | 5.611 |
| BHS18 | 3.098 | 2.571 | 3.726 |
| BCC12-7 | 2.457 | 2.161 | 4.031 |
| BCC12-6 | 2.814 | 2.476 | 3.680 |
| BGHP10 | 3.400 | 2.804 | 3.870 |
| BKPX15 | 3.967 | 3.500 | 5.328 |
| KPT17 | 4.852 | 4.032 | 2.545 |

(c) `normal-clustered`

| Algorithm | $t_{\max}$ | $t_{\text{avg}}$ | $\ell$ |
|---|---|---|---|
| BGS05 | 2.504 | 2.198 | 10.111 |
| LW04 | 2.575 | 2.329 | 4.345 |
| BSX09 | 4.172 | 3.701 | 3.553 |
| KPX10 | 1.500 | 1.450 | 5.319 |
| KX12 | 2.493 | 1.872 | 4.702 |
| BHS18 | 3.090 | 2.538 | 3.288 |
| BCC12-7 | 2.438 | 2.173 | 3.603 |
| BCC12-6 | 2.737 | 2.447 | 3.264 |
| BGHP10 | 3.178 | 2.731 | 3.431 |
| BKPX15 | 4.344 | 3.507 | 4.748 |
| KPT17 | 5.236 | 3.990 | 2.267 |

(d) `normal`

| Algorithm | $t_{\max}$ | $t_{\text{avg}}$ | $\ell$ |
|---|---|---|---|
| BGS05 | 3.000 | 2.812 | 6.911 |
| LW04 | 3.000 | 2.965 | 2.820 |
| BSX09 | 3.000 | 3.000 | 2.455 |
| KPX10 | 1.414 | 1.414 | 3.506 |
| KX12 | 1.414 | 1.414 | 3.480 |
| BHS18 | 1.414 | 1.414 | 2.613 |
| BCC12-7 | 1.414 | 1.414 | 2.850 |
| BCC12-6 | 1.414 | 1.414 | 2.286 |
| BGHP10 | 1.414 | 1.414 | 2.705 |
| BKPX15 | 7.242 | 6.337 | 3.852 |
| KPT17 | 3.000 | 3.000 | 1.915 |

(e) `grid-contiguous`

| Algorithm | $t_{\max}$ | $t_{\text{avg}}$ | $\ell$ |
|---|---|---|---|
| BGS05 | 2.413 | 2.199 | 10.213 |
| LW04 | 2.707 | 2.364 | 4.367 |
| BSX09 | 4.080 | 3.686 | 3.585 |
| KPX10 | 1.508 | 1.452 | 5.382 |
| KX12 | 2.153 | 1.838 | 4.730 |
| BHS18 | 2.905 | 2.555 | 3.318 |
| BCC12-7 | 2.349 | 2.126 | 3.637 |
| BCC12-6 | 2.698 | 2.441 | 3.300 |
| BGHP10 | 3.171 | 2.807 | 3.431 |
| BKPX15 | 4.056 | 3.496 | 4.825 |
| KPT17 | 4.777 | 4.036 | 2.261 |

(f) `grid-random`

| Algorithm | $t_{\max}$ | $t_{\text{avg}}$ | $\ell$ |
|---|---|---|---|
| BGS05 | 2.490 | 2.185 | 10.375 |
| LW04 | 2.735 | 2.344 | 4.445 |
| BSX09 | 4.294 | 3.668 | 3.675 |
| KPX10 | 1.557 | 1.522 | 5.524 |
| KX12 | 2.078 | 1.862 | 4.833 |
| BHS18 | 3.286 | 2.559 | 3.351 |
| BCC12-7 | 2.574 | 2.162 | 3.661 |
| BCC12-6 | 2.922 | 2.466 | 3.321 |
| BGHP10 | 3.077 | 2.756 | 6.154 |
| BKPX15 | 4.226 | 3.527 | 7.273 |
| KPT17 | 4.637 | 4.002 | 3.544 |

(g) `annulus`

| Algorithm | $t_{\max}$ | $t_{\text{avg}}$ | $\ell$ |
|---|---|---|---|
| BGS05 | 2.452 | 2.209 | 12.119 |
| LW04 | 2.673 | 2.323 | 5.114 |
| BSX09 | 4.077 | 3.716 | 4.250 |
| KPX10 | 1.512 | 1.451 | 6.521 |
| KX12 | 2.552 | 1.878 | 5.326 |
| BHS18 | 3.147 | 2.587 | 3.686 |
| BCC12-7 | 2.352 | 2.149 | 3.949 |
| BCC12-6 | 2.803 | 2.483 | 3.608 |
| BGHP10 | 3.309 | 2.773 | 4.398 |
| BKPX15 | 6.833 | 3.615 | 5.806 |
| KPT17 | 4.867 | 4.049 | 2.871 |

(h) `galaxy`

Table 4.2: Stretch-factor and lightness comparisons of the spanners generated by the eleven algorithms.

| Pointset | $n$ | BGS05 | LW04 | BSX09 | KPX10 | KX12 | BHS18 | BCC12-7 | BCC12-6 | BGHP10 | BKPX15 | KPT17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| burma | 33708 | 2.888 | 0.711 | 0.627 | 0.599 | 0.832 | 1.101 | 1.569 | 1.549 | 173.528 | 5.885 | 173.447 |
| birch3 | 99801 | 9.700 | 2.459 | 2.139 | 2.117 | 3.001 | 4.458 | 5.188 | 5.079 | 640.619 | 20.003 | 641.839 |
| mona-lisa | 100000 | 8.527 | 2.450 | 2.127 | 2.236 | 3.189 | 4.960 | 5.863 | 5.588 | 704.980 | 27.988 | 706.000 |
| KDDCU2D | 104297 | 10.024 | 2.579 | 2.252 | 2.250 | 3.146 | 4.837 | 5.569 | 5.460 | 811.921 | 21.424 | 812.665 |
| usa | 115475 | 10.575 | 2.878 | 2.497 | 2.521 | 3.559 | 5.600 | 6.426 | 6.258 | 1033.99 | 35.769 | 1035.64 |
| europe | 168896 | 15.516 | 4.520 | 3.985 | 3.968 | 5.685 | 8.386 | 9.402 | 9.230 | 1494.69 | 39.133 | 1497.09 |
| wiki | 317695 | 35.374 | 9.041 | 7.687 | 8.004 | 11.432 | 17.274 | 18.105 | 17.794 | 3507.06 | 98.975 | 3510.20 |
| vsli | 744710 | 81.461 | 21.655 | 19.457 | 19.693 | 28.165 | 46.630 | 47.297 | 46.183 | 13296.6 | 509.795 | 13312.2 |
| china | 808693 | 91.243 | 24.078 | 21.642 | 21.775 | 30.793 | 52.637 | 54.367 | 53.029 | 17028.4 | 451.383 | 17057.3 |
| uber | 1381253 | 170.493 | 42.199 | 38.227 | 38.268 | 54.486 | 89.577 | 94.444 | 92.359 | 29584.6 | 1689.042 | 29633.1 |
| world | 1904711 | 259.147 | 57.657 | 52.078 | 52.337 | 74.570 | 125.078 | 128.316 | 125.787 | 58707.7 | 1358.70 | 58780.3 |

Table 4.3: Average execution time (in seconds).

| Pointset | $n$ | BGS05 | LW04 | BSX09 | KPX10 | KX12 | BHS18 | BCC12-7 | BCC12-6 | BGHP10 | BKPX15 | KPT17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| burma | 33708 | 11 | 10 | 8 | 14 | 11 | 6 | 7 | 6 | 6 | 4 | 4 |
| birch3 | 99801 | 13 | 10 | 8 | 13 | 11 | 6 | 7 | 6 | 6 | 4 | 4 |
| mona-lisa | 100000 | 11 | 8 | 8 | 12 | 10 | 7 | 7 | 7 | 6 | 4 | 4 |
| KDDCU2D | 104297 | 13 | 10 | 8 | 14 | 10 | 6 | 7 | 7 | 6 | 4 | 4 |
| usa | 115475 | 12 | 10 | 8 | 14 | 11 | 7 | 7 | 7 | 6 | 4 | 4 |
| europe | 168896 | 12 | 10 | 8 | 13 | 10 | 6 | 7 | 6 | 6 | 4 | 4 |
| wiki | 317695 | 14 | 10 | 9 | 14 | 11 | 7 | 7 | 7 | 6 | 4 | 4 |
| vsli | 744710 | 15 | 11 | 9 | 14 | 10 | 7 | 7 | 7 | 6 | 4 | 4 |
| china | 808693 | 13 | 11 | 9 | 14 | 11 | 7 | 7 | 6 | 6 | 4 | 4 |
| uber | 1381253 | 13 | 11 | 9 | 14 | 10 | 7 | 7 | 6 | 6 | 4 | 4 |
| world | 1904711 | 14 | 11 | 9 | 14 | 11 | 7 | 7 | 7 | 6 | 4 | 4 |

Table 4.4: Degree of the spanners.

| Pointset | $n$ | BGS05 | LW04 | BSX09 | KPX10 | KX12 | BHS18 | BCC12-7 | BCC12-6 | BGHP10 | BKPX15 | KPT17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| burma | 33708 | 5.761 | 5.192 | 4.454 | 5.994 | 5.512 | 4.166 | 4.292 | 4.068 | 4.347 | 3.346 | 3.187 |
| birch3 | 99801 | 5.745 | 5.149 | 4.417 | 5.995 | 5.429 | 4.081 | 4.287 | 3.970 | 4.222 | 3.327 | 3.130 |
| mona-lisa | 100000 | 5.938 | 5.596 | 4.617 | 5.996 | 5.981 | 5.259 | 5.741 | 5.165 | 5.434 | 3.572 | 3.613 |
| KDDCU2D | 104297 | 5.720 | 5.127 | 4.399 | 5.985 | 5.390 | 4.047 | 4.294 | 4.015 | 4.216 | 3.325 | 3.122 |
| usa | 115475 | 5.761 | 5.208 | 4.447 | 5.993 | 5.529 | 4.248 | 4.493 | 4.132 | 4.398 | 3.366 | 3.211 |
| europe | 168896 | 5.745 | 5.160 | 4.427 | 5.997 | 5.438 | 4.090 | 4.310 | 3.991 | 4.234 | 3.325 | 3.135 |
| wiki | 317695 | 5.679 | 5.061 | 4.379 | 5.987 | 5.333 | 3.931 | 4.016 | 3.732 | 4.070 | 3.308 | 3.016 |
| vsli | 744710 | 5.749 | 5.152 | 4.416 | 5.994 | 5.438 | 4.096 | 4.334 | 4.007 | 4.277 | 3.316 | 3.176 |
| china | 808693 | 5.770 | 5.209 | 4.437 | 5.996 | 5.519 | 4.245 | 4.506 | 4.154 | 4.368 | 3.344 | 3.208 |
| uber | 1381253 | 5.742 | 5.147 | 4.394 | 5.996 | 5.437 | 4.086 | 4.288 | 3.968 | 4.232 | 3.326 | 3.130 |
| world | 1904711 | 5.748 | 5.171 | 4.438 | 5.991 | 5.489 | 4.151 | 4.371 | 4.020 | 4.318 | 3.344 | 3.168 |

Table 4.5: Average degree per vertex.

| Pointset | $n$ | BGS05 | LW04 | BSX09 | KPX10 | KX12 | BHS18 | BCC12-7 | BCC12-6 | BGHP10 | BKPX15 | KPT17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| burma | 33708 | 2.414 | 2.414 | 3.681 | 1.482 | 1.738 | 2.856 | 2.156 | 2.162 | 3.161 | 4.404 | 4.409 |
| birch3 | 99801 | 2.233 | 2.234 | 3.520 | 1.481 | 1.923 | 2.719 | 2.318 | 2.460 | 2.933 | 3.624 | 4.102 |
| mona-lisa | 100000 | 2.523 | 2.237 | 3.373 | 1.413 | 1.609 | 2.872 | 1.778 | 2.278 | 2.872 | 4.190 | 3.768 |
| KDDCU2D | 104297 | 2.211 | 2.435 | 3.953 | 1.492 | 2.068 | 2.937 | 2.174 | 2.603 | 2.937 | 4.299 | 4.218 |
| usa | 115475 | 2.300 | 2.351 | 3.564 | 1.480 | 2.038 | 2.765 | 2.241 | 2.576 | 3.430 | 3.740 | 4.455 |
| europe | 168896 | 2.245 | 2.343 | 4.072 | 1.459 | 1.840 | 2.745 | 2.310 | 2.659 | 2.981 | 4.081 | 4.121 |
| wiki | 317695 | 2.408 | 2.421 | 3.926 | 1.458 | 1.978 | 2.757 | 2.350 | 2.610 | 3.222 | 4.353 | 4.017 |
| vsli | 744710 | 2.468 | 2.999 | 3.650 | 1.471 | 1.970 | 2.942 | 2.355 | 2.263 | 3.521 | 11.535 | 5.472 |
| china | 808693 | 2.478 | 2.421 | 4.082 | 1.511 | 2.055 | 2.731 | 2.237 | 2.711 | 2.981 | 4.061 | 4.506 |
| uber | 1381253 | 2.535 | 2.418 | 3.987 | 1.485 | 2.204 | 2.902 | 2.407 | 2.816 | 3.073 | 27.929 | 4.966 |
| world | 1904711 | 2.989 | 2.961 | 4.228 | 1.522 | 1.997 | 3.056 | 2.357 | 2.657 | 3.545 | 6.140 | 5.422 |

Table 4.6: Stretch factor of the spanners.

| Pointset | $n$ | BGS05 | LW04 | BSX09 | KPX10 | KX12 | BHS18 | BCC12-7 | BCC12-6 | BGHP10 | BKPX15 | KPT17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| burma | 33708 | 10.755 | 4.538 | 3.768 | 5.672 | 4.922 | 3.374 | 3.609 | 3.365 | 3.620 | 5.048 | 2.345 |
| birch3 | 99801 | 11.008 | 4.660 | 3.845 | 5.805 | 4.989 | 3.453 | 3.770 | 3.434 | 3.699 | 5.124 | 2.426 |
| mona-lisa | 100000 | 7.070 | 3.259 | 2.656 | 3.574 | 3.542 | 3.012 | 3.326 | 2.934 | 3.147 | 3.934 | 1.994 |
| KDDCU2D | 104297 | 10.576 | 4.491 | 3.719 | 5.605 | 4.830 | 3.316 | 3.670 | 3.355 | 3.511 | 4.954 | 2.311 |
| usa | 115475 | 10.264 | 4.427 | 3.663 | 5.431 | 4.753 | 3.336 | 3.642 | 3.305 | 3.602 | 4.935 | 2.336 |
| europe | 168896 | 10.136 | 4.365 | 3.593 | 5.395 | 4.736 | 3.274 | 3.588 | 3.233 | 3.428 | 4.746 | 2.254 |
| wiki | 317695 | 12.137 | 5.087 | 4.227 | 6.555 | 5.385 | 3.607 | 3.877 | 3.525 | 3.860 | 5.477 | 2.481 |
| vsli | 744710 | 11.344 | 4.850 | 4.024 | 5.989 | 5.110 | 3.521 | 3.899 | 3.539 | 3.864 | 5.130 | 2.513 |
| china | 808693 | 9.918 | 4.304 | 3.531 | 5.232 | 4.605 | 3.286 | 3.594 | 3.247 | 3.445 | 4.719 | 2.263 |
| uber | 1381253 | 11.225 | 4.497 | 4.291 | 5.900 | 5.424 | 2.797 | 2.843 | 2.888 | 3.015 | 4.584 | 1.861 |
| world | 1904711 | 11.145 | 4.744 | 3.923 | 5.917 | 5.003 | 3.476 | 3.777 | 3.432 | 3.967 | 5.272 | 2.541 |

Table 4.7: Lightness of the spanners.
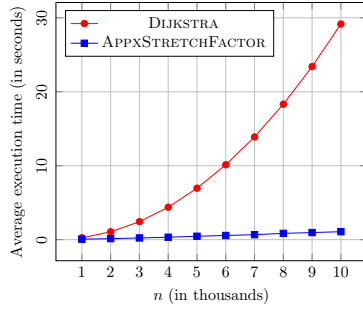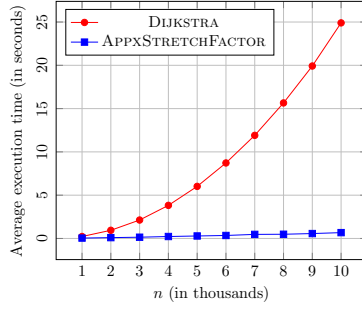
(a) `uni-square`
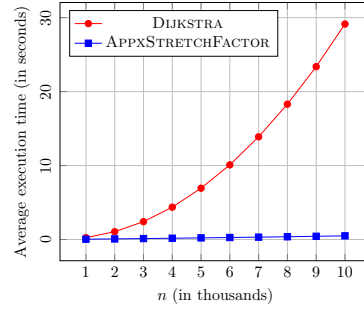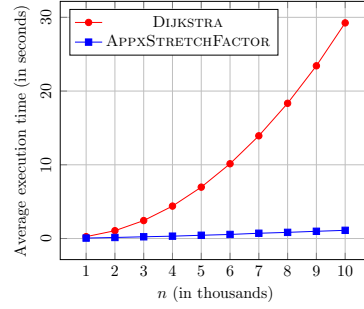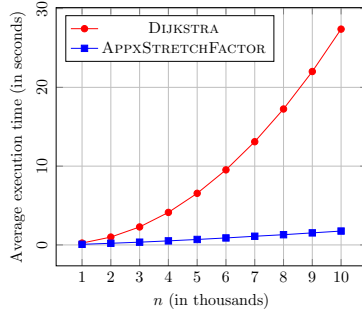
(b) `uni-disk`
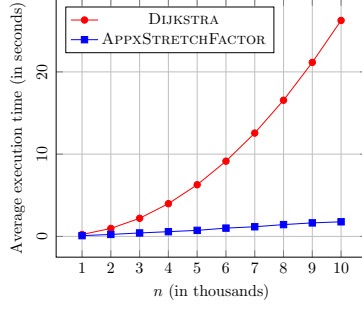
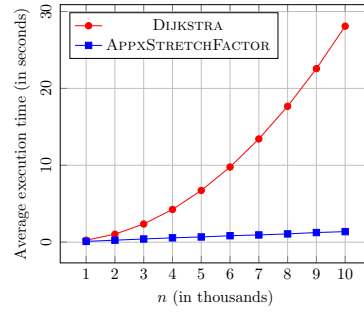(c) `normal-clustered`
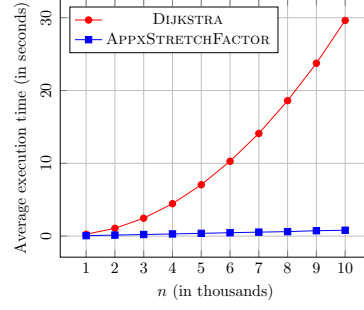
(d) `normal`

(e) `grid-contiguous`

(f) `grid-random`

(g) `annulus`

(h) `galaxy`

Figure 4.3: Runtime comparison: Dijkstra (run from every vertex) vs APPXSTRETCHFACTOR. For every value of $n$, we have used $11 \cdot 5 = 55$ spanner samples since there are 11 algorithms and 5 pointsets were generated for that value of $n$ using the same distribution.
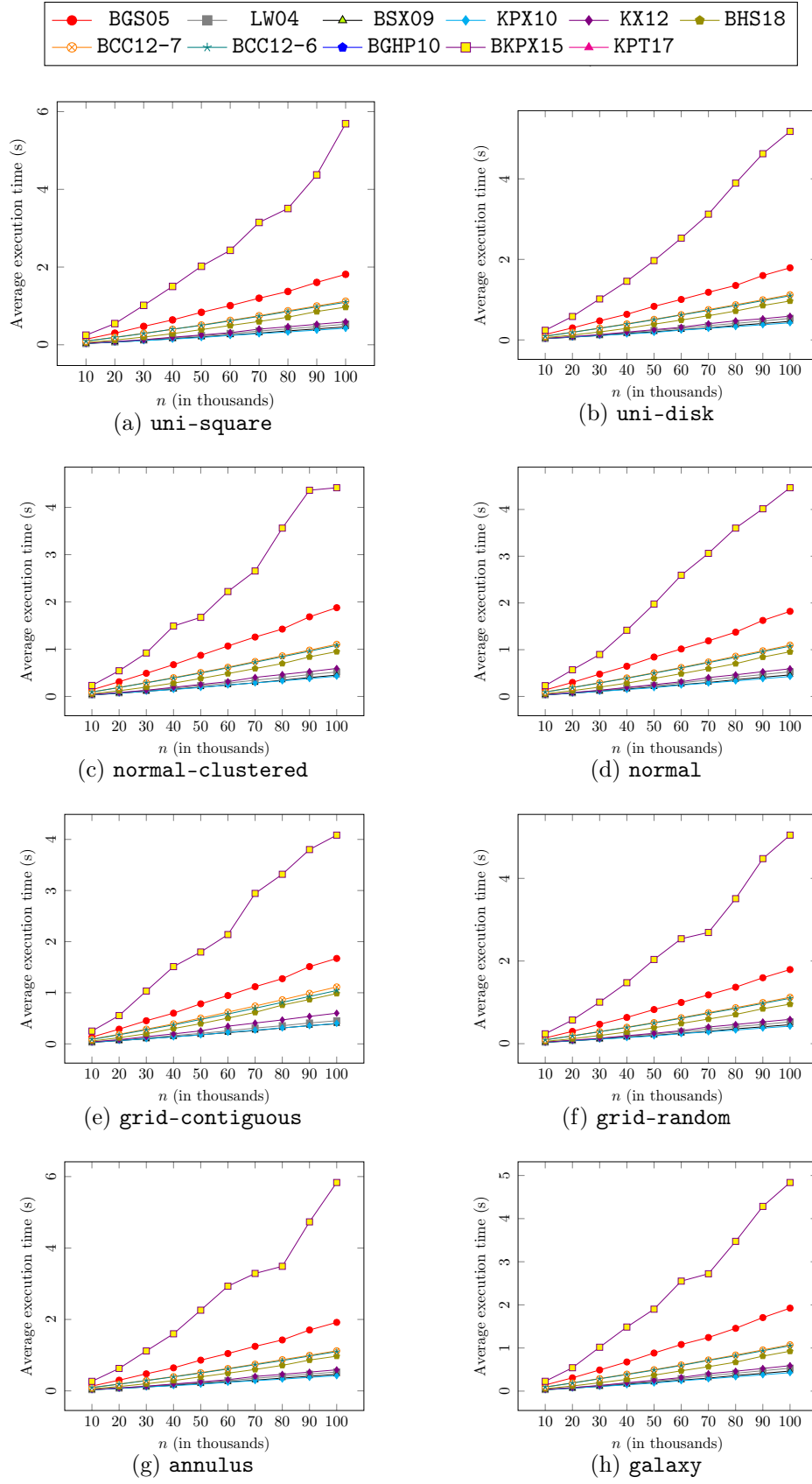
Figure 4.4: Runtime comparisons of the nine algorithms (`BGHP10` and `KPT17` are excluded).

# CHAPTER 5

## Conclusions

Since there are various ways (speed, degree, stretch factor, lightness) to judge the eleven algorithms, it is hard to declare the winner(s). So, based on our experimental observations, we come to the following conclusions (are our recommendations as well):

- If speedy performance is the main concern, we recommend using `KPX10`, `BSX09`, `LW04`, or `KX12`.

- When it comes to minimization of degree, we recommend using `BCC12-7` or `BHS18` since they produce spanners of reasonable degrees in practice. If degree-4 spanners are desired, we recommend using `BKPX15` since `KPT17` is much slower in practice.

- In terms of stretch factor, we find the `KPX10` as the clear winner. This is particularly important in the study of geometric spanners since not much is known about fast construction of low stretch factor spanners ($t \approx 1.6$) in the plane having at most $3n$ edges. However, the spanners produced by it have higher degrees compared to the ones produced by some of the other algorithms such as `BCC12` and `BHS18`.

- In our experiments, `KPT17` produced spanners with the lowest lightnesses. But in practice, we found it to be very slow compared to the other algorithms except for `BGHP10` (which is as slow as `KPT17`). If degree-4 spanners is not a requirement, we recommend using `BHS18` or `BCC12-7` since they produced spanners of reasonable lightness (less than 4 most of the times).

## REFERENCES

[1] www.math.uwaterloo.ca/tsp/.

[2] Pankaj K Agarwal, Rolf Klein, Christian Knauer, Stefan Langerman, Pat Morin, Micha Sharir, and Michael Soss. Computing the detour and spanning ratio of paths, trees, and cycles in 2d and 3d. Discrete & Computational Geometry, 39(1):17–37, 2008.

[3] Fred Anderson, Anirban Ghosh, Matthew Graham, Lucas Mougeot, and David Wisnosky. An interactive tool for experimenting with bounded-degree plane geometric spanners (media exposition). In 37th International Symposium on Computational Geometry (SoCG 2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[4] Davood Bakhshesh and Mohammad Farshi. A degree 3 plane 5.19-spanner for points in convex position. In CCCG, pages 226–232, 2020.

[5] Jon Louis Bentley. K-d trees for semidynamic point sets. In Proceedings of the Sixth Annual Symposium on Computational Geometry, pages 187–197, 1990.

[6] Ahmad Biniaz. Plane hop spanners for unit disk graphs: Simpler and better. Comput. Geom., 89:101622, 2020.

[7] Ahmad Biniaz, Prosenjit Bose, Jean-Lou De Carufel, Cyril Gavoille, Anil Maheshwari, and Michiel Smid. Towards plane spanners of degree 3. Journal of Computational Geometry, 8(1):11–31, 2017.

[8] Nicolas Bonichon, Cyril Gavoille, Nicolas Hanusse, and David Ilcinkas. Connections between theta-graphs, Delaunay triangulations, and orthogonal surfaces. In International

Workshop on Graph-Theoretic Concepts in Computer Science, pages 266–278. Springer, 2010.

[9] Nicolas Bonichon, Cyril Gavoille, Nicolas Hanusse, and Ljubomir Perković. Plane spanners of maximum degree six. In International Colloquium on Automata, Languages, and Programming, pages 19–30. Springer, 2010.

[10] Nicolas Bonichon, Cyril Gavoille, Nicolas Hanusse, and Ljubomir Perković. The stretch factor of $L_1$-and $L_\infty$-Delaunay triangulations. In European Symposium on Algorithms, pages 205–216. Springer, 2012.

[11] Nicolas Bonichon, Iyad Kanj, Ljubomir Perković, and Ge Xia. There are plane spanners of degree 4 and moderate stretch factor. Discrete & Computational Geometry, 53(3):514–546, 2015.

[12] Prosenjit Bose, Paz Carmi, and Lilach Chaitman-Yerushalmi. On bounded degree plane strong geometric spanners. Journal of Discrete Algorithms, 15:16–31, 2012.

[13] Prosenjit Bose, Joachim Gudmundsson, and Michiel Smid. Constructing plane spanners of bounded degree and low weight. Algorithmica, 42(3-4):249–264, 2005.

[14] Prosenjit Bose, Darryl Hill, and Michiel Smid. Improved spanning ratio for low degree plane spanners. Algorithmica, 80(3):935–976, 2018.

[15] Prosenjit Bose and Michiel Smid. On plane geometric spanners: A survey and open problems. Computational Geometry, 46(7):818–830, 2013.

[16] Prosenjit Bose, Michiel Smid, and Daming Xu. Delaunay and diamond triangulations contain spanners of bounded degree. International Journal of Computational Geometry & Applications, 19(02):119–140, 2009.

[17] Norbert Bus, Nabil H Mustafa, and Saurabh Ray. Practical and efficient algorithms for the geometric hitting set problem. Discrete Applied Mathematics, 240:25–32, 2018.

[18] 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 (JACM), 42(1):67–90, 1995.

[19] Nicolas Catusse, Victor Chepoi, and Yann Vaxès. Planar hop spanners for unit disk graphs. In International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics, pages 16–30. Springer, 2010.

[20] Siu-Wing Cheng, Christian Knauer, Stefan Langerman, and Michiel Smid. Approximating the average stretch factor of geometric graphs. Journal of Computational Geometry, 3(1):132–153, 2012.

[21] L Paul Chew. There is a planar graph almost as good as the complete graph. In Proceedings of the Second Annual Symposium on Computational Geometry, 1986.

[22] L Paul Chew. There are planar graphs almost as good as the complete graph. Journal of Computer and System Sciences, 39(2):205–219, 1989.

[23] Gautam Das and Paul J Heffernan. Constructing degree-3 spanners with other sparseness properties. International Journal of Foundations of Computer Science, 7(02):121–135, 1996.

[24] Adrian Dumitrescu and Anirban Ghosh. Lattice spanners of low degree. Discrete Mathematics, Algorithms and Applications, 8(03):1650051, 2016.

[25] Adrian Dumitrescu and Anirban Ghosh. Lower bounds on the dilation of plane spanners. International Journal of Computational Geometry & Applications, 26(02):89–110, 2016.

[26] Adrian Dumitrescu, Anirban Ghosh, and Csaba D Tóth. Sparse hop spanners for unit disk graphs. In 31st International Symposium on Algorithms and Computation (ISAAC 2020). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[27] Mohammad Farshi and Joachim Gudmundsson. Experimental study of geometric $t$-spanners. Journal of Experimental Algorithmics (JEA), 14:3, 2009.

[28] Greg N Federickson. Fast algorithms for shortest paths in planar graphs, with applications. SIAM Journal on Computing, 16(6):1004–1022, 1987.

[29] Itinerant Games. A 2d procedural galaxy with `C++`, Mar 2014. URL: `https://itinerantgames.tumblr.com/post/78592276402/a-2d-procedural-galaxy-with-c`.

[30] Anirban Ghosh, Brian Hicks, and Ronald Shevchenko. Unit disk cover for massive point sets. In Special Event on Analysis of Experimental Algorithms. Springer (to appear), 2019.

[31] Anirban Ghosh, Brian Hicks, and Ronald Shevchenko. Unit disk cover for massive point sets. In International Symposium on Experimental Algorithms, pages 142–157. Springer, 2019.

[32] Iyad Kanj, Ljubomir Perkovic, and Duru Türkoğlu. Degree four plane spanners: Simpler and better. Journal of Computational Geometry, 8(2):3–31, 2017.

[33] Iyad A Kanj, Ljubomir Perković, and Ge Xia. On spanners and lightweight spanners of geometric graphs. SIAM Journal on Computing, 39(6):2132–2161, 2010.

[34] Iyad A Kanj and Ge Xia. Improved local algorithms for spanner construction. Theoretical Computer Science, 453:54–64, 2012.

[35] Rolf Klein, Martin Kutz, and Rainer Penninger. Most finite point sets in the plane have dilation > 1. Discrete & Computational Geometry, 53(1):80–106, 2015.

[36] Xiang-Yang Li and Yu Wang. Efficient construction of low weighted bounded degree planar spanner. International Journal of Computational Geometry & Applications, 14(01n02):69–84, 2004.

[37] Wolfgang Mulzer. Minimum dilation triangulations for the regular $n$-gon. Master's thesis, Freie Universität Berlin, Germany, 2004.

[38] Giri Narasimhan and Michiel Smid. Approximating the stretch factor of Euclidean graphs. SIAM Journal on Computing, 30(3):978–989, 2000.

[39] Giri Narasimhan and Michiel Smid. Geometric spanner networks. Cambridge University Press, 2007.

[40] Giri Narasimhan and Martin Zachariasen. Geometric minimum spanning trees via well-separated pair decompositions. Journal of Experimental Algorithmics (JEA), 6:6–es, 2001.

[41] The CGAL Project. CGAL User and Reference Manual. CGAL Editorial Board, 5.3 edition, 2021. URL: https://doc.cgal.org/5.3/Manual/packages.html.

[42] Csaba D Toth, Joseph O'Rourke, and Jacob E Goodman. Handbook of Discrete and Computational Geometry. Chapman and Hall/CRC, 2017.

[43] Christian Wulff-Nilsen. Computing the maximum detour of a plane geometric graph in subquadratic time. Journal of Computational Geometry, 1(1):101–122, 2010.

[44] Ge Xia. The stretch factor of the Delaunay triangulation is less than 1.998. SIAM Journal on Computing, 42(4):1620–1659, 2013.
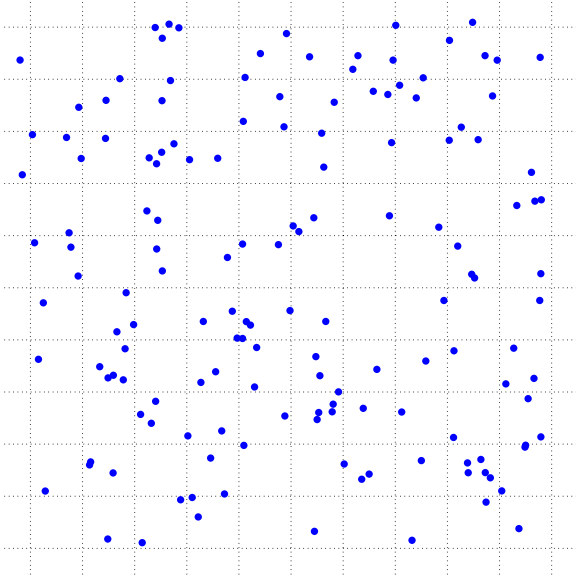
Figure A.1: A 150-element pointset, drawn randomly from a square.
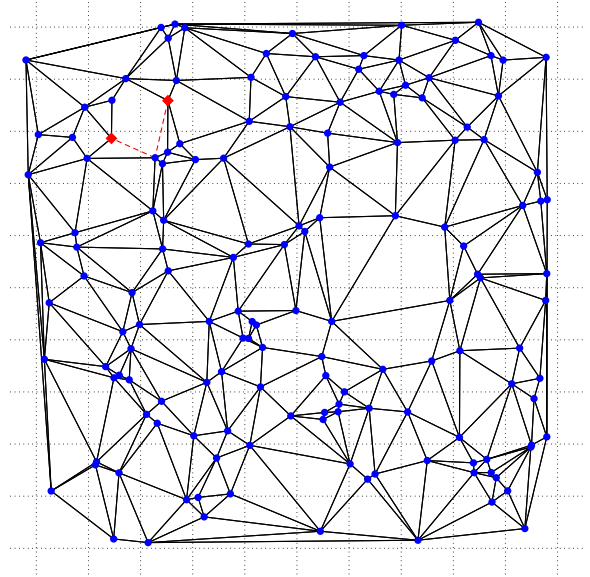


Figure A.2: The spanner generated by `BGS05` on the pointset shown in Fig. A.1; degree: 8, stretch factor: 1.565763
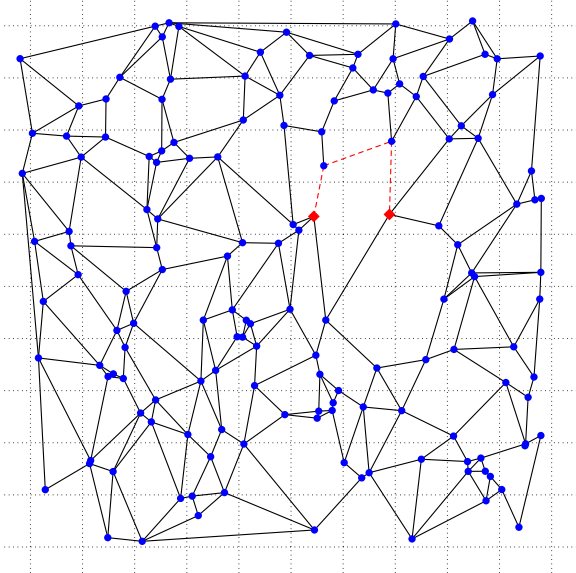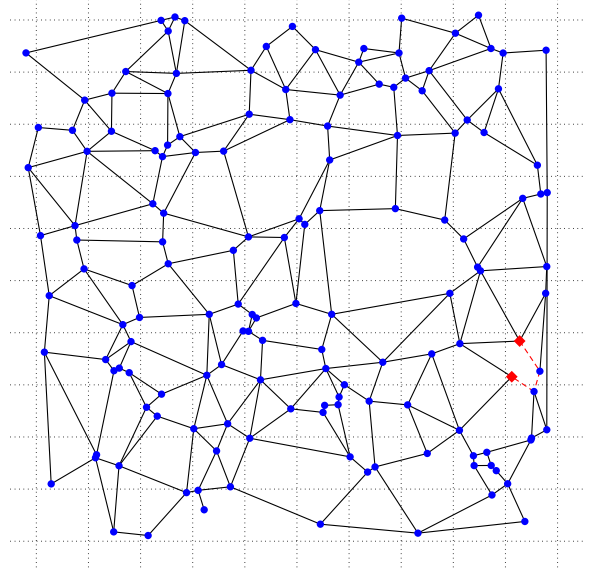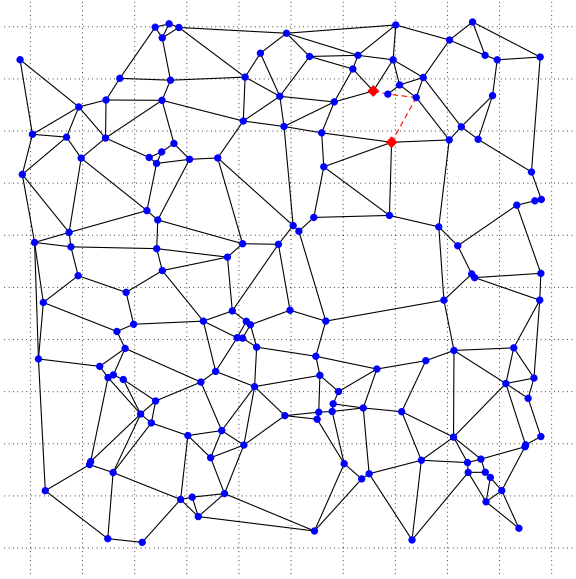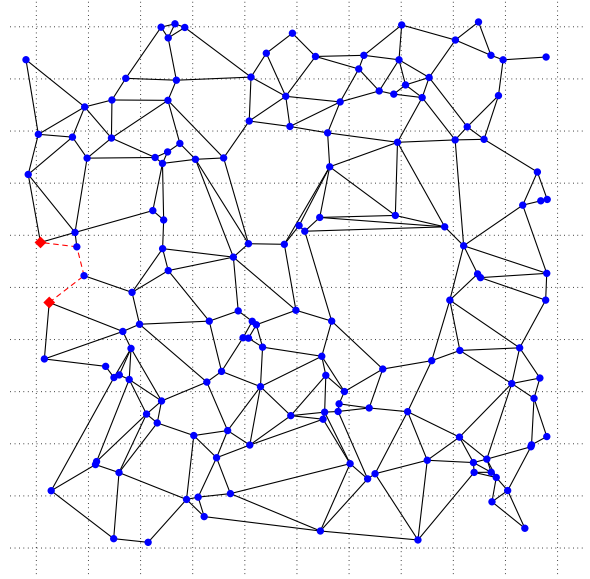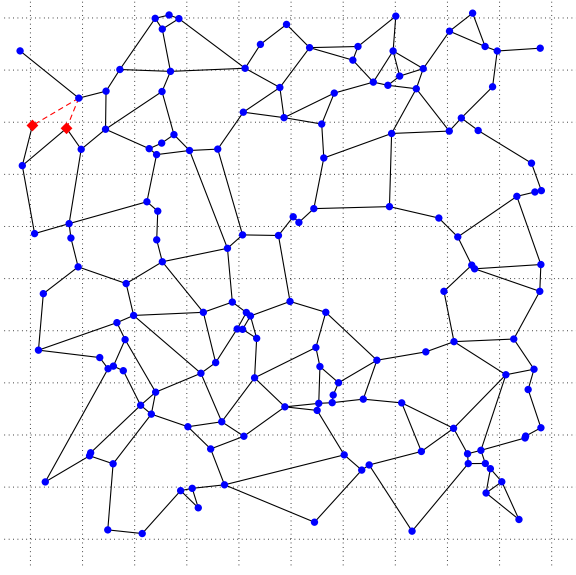


Figure A.3: The spanner generated by `LW04` on the pointset shown in Fig. A.1; degree: 6, stretch factor: 2.602559
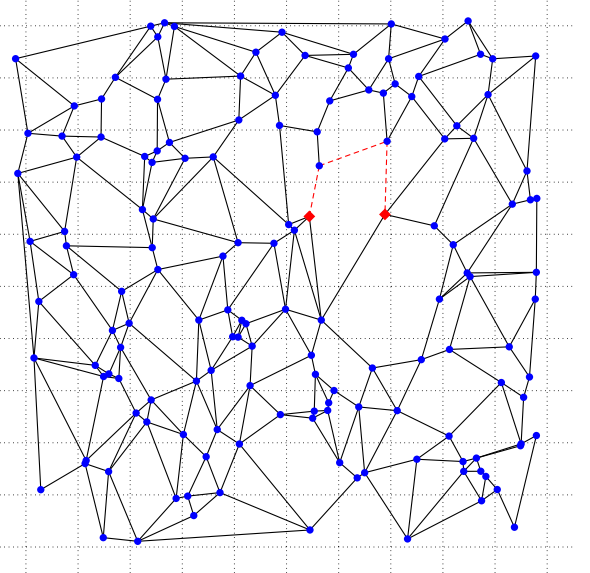


Figure A.4: The spanner generated by `BSX09` on the pointset shown in Fig. A.1; degree: 6, stretch factor: 2.602559
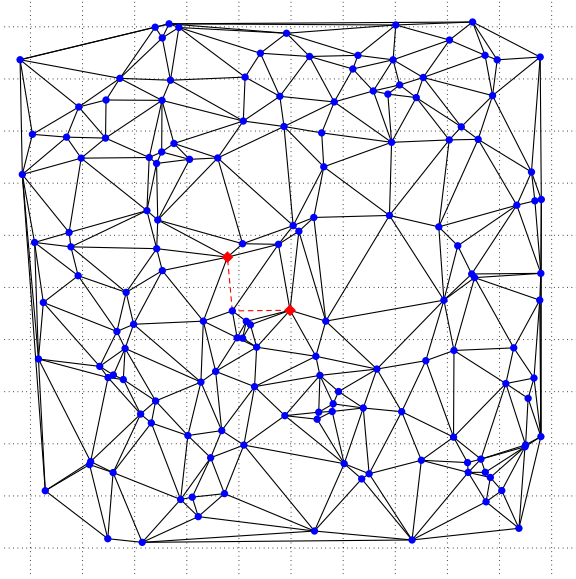
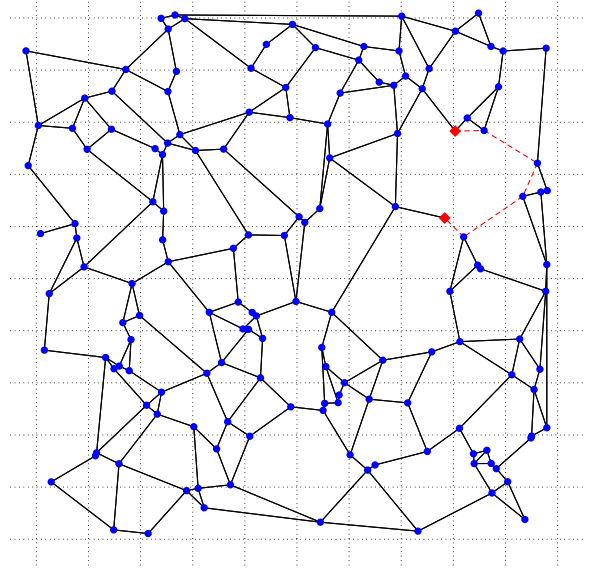Figure A.5: The spanner generated by `KPX10` on the pointset shown in Fig. A.1; degree: 9, stretch factor: 1.360771
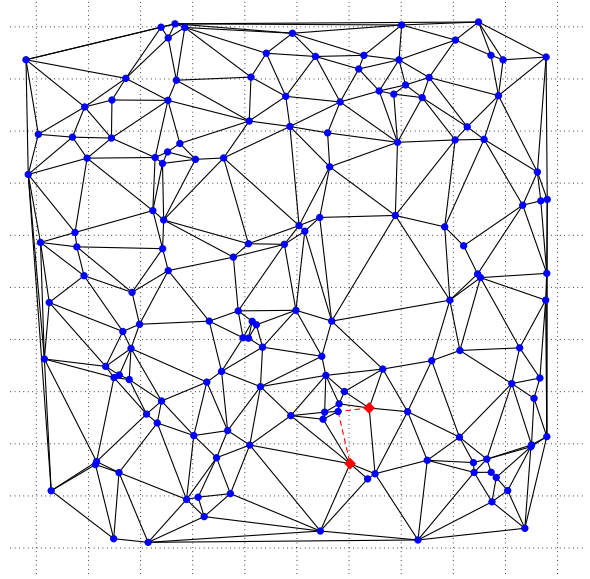


Figure A.6: The spanner generated by `KX12` on the pointset shown in Fig. A.1; degree: 8, stretch factor: 1.440861
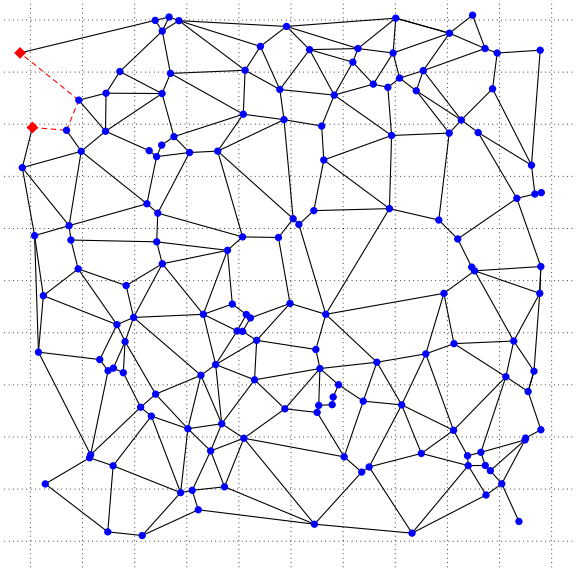


Figure A.7: The spanner generated by `BHS18` on the pointset shown in Fig. A.1; degree: 6, stretch factor: 1.879749



Figure A.8: The spanner generated by `BCC12-7` on the pointset shown in Fig. A.1; degree: 6, stretch factor: 2.302473

Figure A.9: The spanner generated by `BCC12-6` on the pointset shown in Fig. A.1; degree: 6, stretch factor: 1.735716



Figure A.10: The spanner generated by `BGHP10` on the pointset shown in Fig. A.1; degree: 6, stretch factor: 1.817045



Figure A.11: The spanner generated by `BKPX15` on the pointset shown in Fig. A.1; degree: 4, stretch factor: 2.525204



Figure A.12: The spanner generated by `KPT17` on the pointset shown in Fig. A.1; degree: 4, stretch factor: 2.582846

# APPENDIX B

## A COUNTEREXAMPLE FOR BCC12-6

In the following, we present an 13-element pointset on which BCC12-6 fails to construct a degree-6 plane spanner. Refer to Fig. B.1 for the pointset.
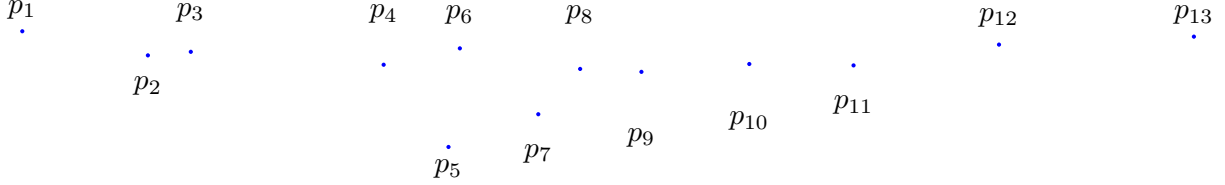


Figure B.1: A set $P$ of 13 points $p_1, \ldots, p_{13}$. $p_1$: $(-4.98845, 0.22414)$, $p_2$: $(-4.23759, 0.08)$, $p_3$: $(-3.98106, 0.10125)$, $p_4$: $(-2.82831, 0.02396)$, $p_5$: $(-2.44066, -0.46761)$, $p_6$: $(-2.37275, 0.12191)$, $p_7$: $(-1.90395, -0.27187)$, $p_8$: $(-1.65373, -0.00109)$, $p_9$: $(-1.28739, -0.01854)$, $p_{10}$: $(-0.642516, 0.02836)$, $p_{11}$: $(-0.019359, 0.02)$, $p_{12}$: $(0.850154, 0.14431)$, $p_{13}$: $(2.01517, 0.19194)$

First, BCC12-6 creates the $L_2$-Delaunay triangulation of $P$ and initializes 7 cones around every $p_i$, oriented such that the shortest edge incident on $p_i$ falls on a boundary. See Figs. B.2 and B.3.
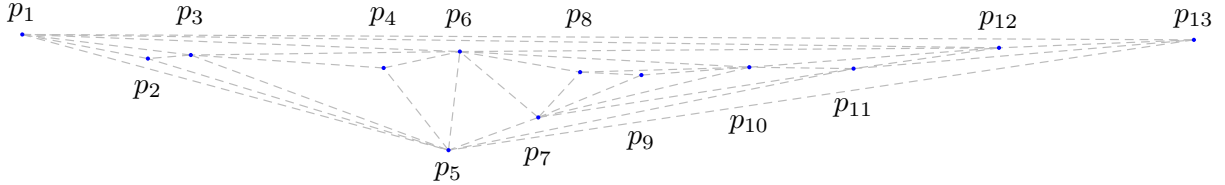


Figure B.2: The $L_2$-Delaunay triangulation of $P$.
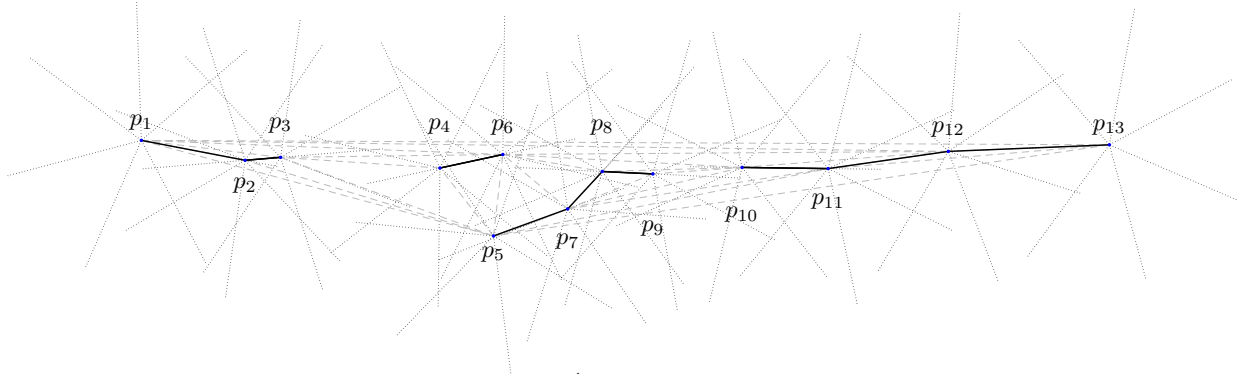
Figure B.3: The cones (dotted) of each point in $P$ with $\alpha = 2\pi/7$, oriented by the shortest edge incident on that point (bold).

Next, in Fig B.4, we show the edges added by the main portion of the algorithm (excluding the edges added by $\mathtt{Wedge}_6$ calls). Only $\mathtt{Wedge}_6(p_1, p_2)$ and $\mathtt{Wedge}_6(p_{12}, p_{11})$ calls add new edges to $E^*$ and thus to the final spanner as well. The former call adds the two edges $p_3 p_6, p_6 p_{12}$ (see Fig. B.5) and the later call adds the edge $p_6 p_{10}$ (see Fig. B.6). The final spanner is shown in Fig. B.7. Note that $p_6$ has degree 7 in the spanner which violates the degree requirement of the spanners produced by $\mathtt{BCC12\text{-}6}$.
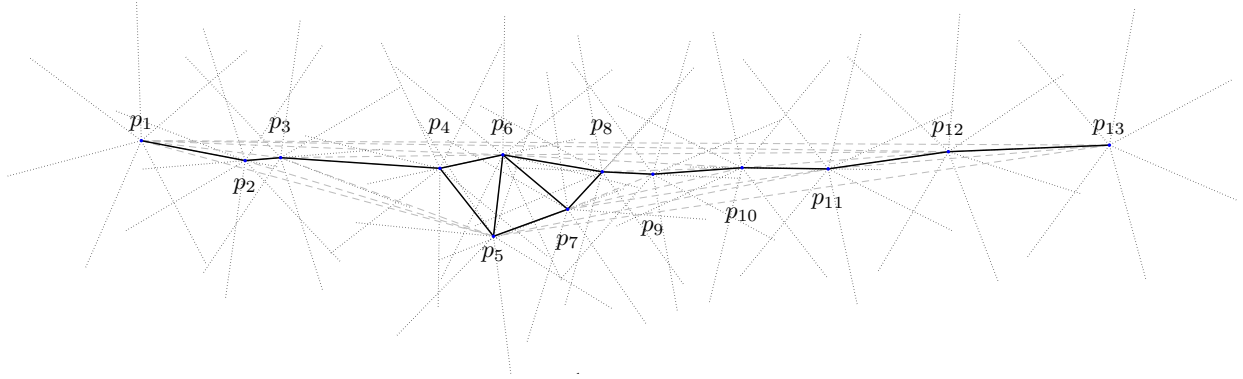


Figure B.4: Edges added by the main portion of $\mathtt{BCC12}$ (excluding calls to subroutine $\mathtt{Wedge}_6$).
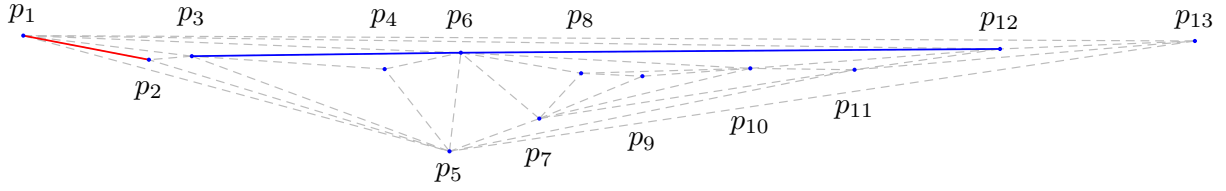
Figure B.5: The edge $p_1p_2$ (shown in red) is added during the main portion of the algorithm and the call to $\texttt{Wedge}_6(p_1, p_2)$ adds the two blue edges $p_3p_6$ and $p_6p_{12}$.
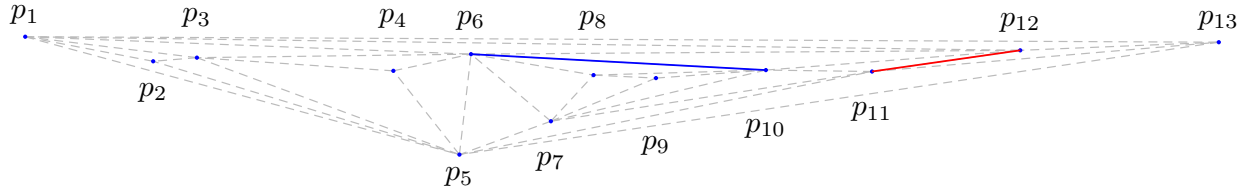


Figure B.6: The edge $p_{12}p_{11}$ (shown in red) is added during the main portion of the algorithm and the call to $\texttt{Wedge}_6(p_{12}, p_{11})$ adds the blue edge $p_6p_{10}$.
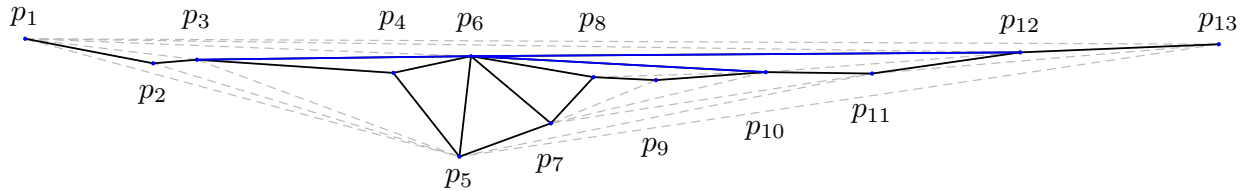


Figure B.7: The resulting graph on $P$ is a degree-7 plane spanner due to $p_6$ whose degree is exactly 7. Note that this graph contains the edges shown in Fig. B.4 alongwith the blue edges shown in Fig. B.5 and B.6.

# VITA

Matthew A. Graham graduated from Clay High School in Green Cove Springs, Florida with the class of 2007 then started at the University of North Florida in Jacksonville shortly afterward, but in March 2009, he enlisted in the United States Navy as an Electronics Technician. Serving just shy of 10 years, Matthew spent his first tour of duty aboard the aircraft carrier USS Ronald Reagan, where he was awarded the Air Traffic Control Technician of the Year for the U.S. Navy and Marine Corps in the year 2014. Subsequently, he spent his final years of military service as an instructor, teaching radar theory and repair, which afforded him the opportunity to return to the college classroom at Pensacola State College and the University of West Florida.

After separating from the military in 2018, Matthew returned to Jacksonville and UNF. In December of 2019, he earned a Bachelor of Science in Psychology with a minor in Computer Science, and, in January of 2020, he entered the Graduate School at UNF to study towards a Master of Science in Computing and Information Sciences with concentration in Computer Science, completed in May 2022 under the supervision of thesis advisor Dr. Anirban Ghosh. During his time as a student at UNF, he worked as a grader for the Introduction to Computer Science class and a graduate research assistant working on algorithm engineering in computational geometry.

Matthew currently works as a software engineer in the broadcast-media industry and is the Treasurer of the Parent-Teacher Association at Ruth N. Upson Elementary School in Jacksonville, Florida.