Bounded-Degree Plane Geometric Spanners in Practice

FREDERICK ANDERSON, ANIRBAN GHOSH, MATTHEW GRAHAM, LUCAS MOUGEOT, and DAVID WISNOSKY, University of North Florida

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, 11 algorithms have been designed with various tradeoffs in degree and stretch-factor. We have implemented these sophisticated spanner 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 for broader uses and future research.

We design and engineer EstimateStretchFactor, a simple practical algorithm, which 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 found that EstimateStretchFactor estimated stretch-factors almost precisely. Further, it gave 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.

CCS Concepts: • Theory of computation → Sparsification and spanners;

Additional Key Words and Phrases: Geometric graph, plane spanner, stretch-factor

ACM Reference format:
Frederick Anderson, Anirban Ghosh, Matthew Graham, Lucas Mougeot, and David Wisnosky. 2023. Bounded-Degree Plane Geometric Spanners in Practice. ACM J. Exp. Algor. 28, 1, Article 1.1 (April 2023), 36 pages. https://doi.org/10.1145/3582497

1 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. Figure 1 presents 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.

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 $T$. Bose et al. [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 bound for the stretch-factor of $L_2$-Delaunay triangulations (the current best known value is $\sigma = 1.998$ due to Xia [45]). 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. [32] improved this stretch-factor upper bound to 20 in their work. A summary of these results is presented in Table 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 the work of Bose and Smid [15, Problem 14] and Toth et al. [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 degree of 3 is necessary to achieve a constant stretch-factor. If the points in $P$ are in convex position, then it is always possible to construct a degree-3 plane geometric spanners (see [3, 7, 32]). From the other direction, lower bounds on the stretch-factors of plane spanners for finite pointsets have been investigated elsewhere [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 the work of Anderson et al. [2].

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 [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 11 algorithms (refer to Table 1) meant to construct bounded-degree plane geometric spanners. Further, 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 heavy floating-point operations needed for various geometric calculations. Experiments reveal their real-world performance. We note that 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 elsewhere [12, 39] for a review of such algorithms. This makes experiments with them even more interesting.

Our Contributions. First, we experimentally compare the aforementioned 11 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 approximately 1.9 million points. For broader uses of these sophisticated algorithms, we share the C++ implementations via GitHub. The comparisons are performed based on their runtime, 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. The findings of our experimental study are presented in Section 5.

Second, in doing experiments with spanners, we found that stretch-factor measurement turns out to be a severe bottleneck when $n$ is large. To address this, we have designed ESTIMATESTRETCH-FACTOR, a fast algorithm that can estimate the stretch-factor of a given spanner (not necessarily plane). In our experiments, we found that it could estimate stretch-factors with high accuracy for the class of geometric spanners dealt with in this work. It was considerably faster than the naive

| Reference      | Degree | Stretch-Factor                                      |
|----------------|--------|----------------------------------------------------|
| Bose et al. [13] | 27     | $\sigma(\pi + 1) \approx 8.3$                      |
| Li and Wang [56] | 23     | $\sigma(1 + \frac{2\pi}{\sqrt{2}}) \approx 6.4$   |
| Bose et al. [16]  | 17     | $\sigma(2 + 2\sqrt{3} + \frac{2\pi}{\sqrt{2}} + 2\pi\sin\frac{\pi}{4}) \approx 23.6$ |
| Kanj et al. [33]  | 14     | $\sigma(1 + \frac{2\pi}{14}\cos(\pi/14)) \approx 2.9$ |
| Kanj and Xia [34] | 11     | $\sigma(\frac{2}{2}\sin(\frac{\pi}{14})\cos(\pi/3)) \approx 5.7$ |
| Bose et al. [14]  | 8      | $\sigma(1 + \frac{2\pi}{6}\cos(\pi/6)) \approx 4.4$ |
| Bose et al. [12]  | 7      | $\sigma(1 + \sqrt{2})^2 \approx 11.6$            |
| Bose et al. [12]  | 6      | $\sigma(1 - \tan(\pi/7)(1+1/\cos(\pi/14)) \approx 81.7$ |
| Bonichon et al. [9] | 6     | $\sqrt{4 + 2\sqrt{2}(19 + 29\sqrt{2})} \approx 156.8$ |
| Bonichon et al. [11] | 4     | $\frac{1}{2}\cos(\pi/6) \approx 6$                |
| Kanj et al. [32]  | 4      | $\frac{1}{2}$                                     |

The best known upper bound of $\sigma = 1.998$ for the stretch-factor of the $L_2$-Delaunay triangulation [45] is used in this table for expressing the stretch-factors.
Dijkstra-based exact stretch-factor measurement algorithm in practice. To our knowledge, no such practical algorithm exists in the literature. Section 3 presents a description of this algorithm.

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, 45] 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 11 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, 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 developed by us (see [2]). To observe variations in spanner construction between the algorithms, see Appendix A.1.

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 $C^u_i$. A triangulation $T$ of a pointset $P$ is said to be an $L_2$-Delaunay triangulation of $P$ if no point in $P$ lies inside the circumcircle of any triangle in $T$. Eight of the 11 algorithms use $L_2$-Delaunay triangulation as the starting point. The remaining 3 algorithms use either $L_∞$ or $TD$-Delaunay triangulations, as described later in this section. In the following, $n$ denotes the size of the input pointset:

- **BGS05: Bose et al. [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 $DT$, then Delaunay edges are carefully added to $Q$. The resulting graph denoted $G'$ is a plane spanner for the vertices of $Q$. Refer to Algorithm 5 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.

```
ALGORITHM 1: CanonicalOrdering(DT)
1 Declare an empty array $\Phi[1, \ldots, n]$;
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 $H \setminus \text{reserved}$ that is adjacent to at most two other
7     vertices on the outer face;
8     $\Phi[u] \leftarrow n - i + 1$;
9     Remove $u$ and all incident edges from $H$;
10 end
11 return $\Phi$;
```
ALGORITHM 2: SpanningGraph(DT)
1 $\Phi[1, \ldots, n] \leftarrow \text{CanonicalOrdering}(DT)$ (Algorithm 1);
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 $u_1, \ldots, u_k$ 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 3: 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 4: 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
9       $s_j, s_k \in E'$;
10     end
11   Divide $\angle s_1 u s_j$ and $\angle s_k u s_m$ into an minimum number of cones with maximum angle $\pi/2$;
12   In each cone, add the shortest edge in $E$ incident upon $u$ to $E'$ and all edges $\{s_\ell, s_{\ell+1}\}$ such
13     that $1 \leq \ell < j$ or $k \leq \ell < m$;
14 end
15 return $E'$;
ALGORITHM 5: BGS05(P)

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

ALGORITHM 6: ReverseLowDegreeOrdering($DT$)

1. Declare an empty array $\Phi[1 \ldots n]$;
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 $H$;
4. return $\Phi$;

ALGORITHM 7: LW04($P, 0 < \alpha \leq \pi/2$)

1. $DT \leftarrow L_2$-DelaunayTriangulation(P);
2. $\Phi[1 \ldots n] \leftarrow$ ReverseLowDegreeOrdering($DT$) (Algorithm 6);
3. $E \leftarrow \emptyset$;
4. foreach $u \in \Phi$ do
   5. if $u$ has unprocessed Delaunay neighbors then
      6. Divide the area surrounding $u$ into sectors delineated by these unprocessed neighbors;
      7. Divide each sector into a minimum number of equal-sized cones $C_{u}^0, C_{u}^1, \ldots$ with angle at most $\alpha$;
      8. foreach $C_{u}^i$ do
         9. Let $v_1, v_2, \ldots, v_m$ be the clockwise-ordered Delaunay neighbors of $u$ in $C_{u}^i$;
         10. Let $v_{\text{closest}}$ be the closest unprocessed neighbor to $u$;
         11. Add edge $\{u, v_{\text{closest}}\}$ to $E$;
         12. Add all edges $\{v_j, v_{j+1}\}$ such that $1 \leq j < m$ to $E$;
   13. Mark $u$ as processed;
4. return $E$;

• LW04: Li and Wang [36]: This algorithm is inspired by 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. Algorithm 7 presents 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 $1.998(1 + \pi/\sqrt{2}) \approx 6.4$ and runs in $O(n \log n)$ time.

• BSX09: Bose et al. [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 8. 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 $u$, $\delta = \lceil \frac{2\pi}{\alpha} \rceil$ 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 neighbors are unprocessed.

**ALGORITHM 8: BSX09 ($0 < \alpha \leq \frac{2\pi}{3}$)**

1. $DT \leftarrow L_2$-DelaunayTriangulation($P$);
2. $\Phi[1...n] \leftarrow \text{ReverseLowDegreeOrdering}(DT)$ (Algorithm 6);
3. $E \leftarrow \emptyset$;
4. foreach $u \in \Phi$ do
   5. if $u$ has unprocessed Delaunay neighbors then
      6. Let $v_{\text{closest}}$ be the closest unprocessed neighbor to $u$;
      7. Add the edge $\{u, v_{\text{closest}}\}$ to $E$;
      8. Divide the area surrounding $u$ into $\lceil \frac{2\pi}{\alpha} \rceil$ non-overlapping cones $C^0_u, C^1_u, \ldots$ such that $v_{\text{closest}}$ is on the boundary between the first and last cones;
      9. foreach $C^i_u$ except the first and last do
         10. if $u$ has unprocessed neighbors in $C^i_u$ then
              11. Let $w$ be the closest unprocessed neighbor to $u$ in the cone;
              12. Add edge $\{u, w\}$ to $E$;
         end
      end
      13. Let $v_0, v_1, \ldots, v_{m-1}$ be the clockwise-ordered neighbors of $u$;
      14. Add all edges $\{v_j, v_{(j+1) \mod m}\}$ to $E$ such that $0 \leq j < m$ and $v_j, v_{(j+1) \mod m}$ are unprocessed;
   end
5. Mark $u$ as processed;
6. return $E$;

- **BGHP10: Bonichon et al. [9]**: This 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 [22] in 1989. 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 9. It uses internally Algorithms 10 through 15. In this algorithm, charge($u, i$) maps vertex $u \in DT$ and a cone $i$ of $u$ to the number of edges charged to the cone, initialized to 0 in the beginning. The algorithm runs in $O(n \log n)$ time, as shown by the authors.

- **KPx10: Kanj et al. [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. Algorithm 16 presents 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 \( 1.998(1 + \frac{2\pi}{14\cos(\pi/14)}) \approx 2.9 \). 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). KPX10 runs in \( O(n \log n) \) time.

**Algorithm 9:** BGHP10(\( P \))

1. **Notations.** Refer to Algorithms 10 through 15 to see how \( i \)-relevant(\( v, u, i \)), \( i \)-distant(\( w, i \)), parent(\( u, i \)), closest(\( u, i \)), first(\( u, i \)), and last(\( u, i \)) are defined.
2. \( DT \leftarrow \) TD-DelaunayTriangulation(\( P \));
3. \( E \leftarrow \emptyset \);
4. **foreach** nonempty cone \( i \) of vertex \( u \in DT \) where \( i \in \{1, 3, 5\} \) **do**
   5. Add edge \( \{u, \text{close}(u, i)\} \) to \( E \);
   6. \( \text{charge}(u, i) \leftarrow \text{charge}(u, i) + 1 \);
   7. \( \text{charge}((\text{close}(u, i), i + 3)) \leftarrow \text{charge}((\text{close}(u, i), i + 3)) + 1 \);
   8. **if** first(\( u, i \)) \( \neq \) closest(\( u, i \)) \( \wedge \) \( i \)-relevant(first(\( u, i \)), \( u, i - 1 \)) **then**
      9. Add edge \( u, \text{first}(u, i) \) to \( E \);
     10. \( \text{charge}(u, i - 1) \leftarrow \text{charge}(u, i - 1) + 1 \);
   **end**
   12. **if** last(\( u, i \)) \( \neq \) closest(\( u, i \)) \( \wedge \) \( i \)-relevant(last(\( u, i \)), \( u, i + 1 \)) **then**
      13. Add edge \( \{u, \text{last}(u, i)\} \) to \( E \);
     14. \( \text{charge}(u, i + 1) \leftarrow \text{charge}(u, i + 1) + 1 \);
   **end**
5. **end**
6. **foreach** cone \( i \) of vertex \( u \in DT \) where \( i \in \{0, 2, 4\} \) such that \( i \)-distant(\( u, i \)) is true **do**
    7. \( v_{\text{next}} \leftarrow \text{first}(u, i + 1) \);
    8. \( v_{\text{prev}} \leftarrow \text{last}(u, i - 1) \);
    9. Add edge \( \{v_{\text{next}}, v_{\text{prev}}\} \) to \( E \);
    10. \( \text{charge}(v_{\text{next}}, i + 1) \leftarrow \text{charge}(v_{\text{next}}, i + 1) + 1 \);
    11. \( \text{charge}(v_{\text{prev}}, i - 1) \leftarrow \text{charge}(v_{\text{prev}}, i - 1) + 1 \);
    12. Let \( v_{\text{remove}} \) be the vertex from \( v_{\text{next}}, v_{\text{prev}} \) where \( \angle(\text{parent}(u, i), u, v_{\text{remove}}) \) is maximized;
    13. Remove edge \( \{u, v_{\text{remove}}\} \) from \( E \);
    14. \( \text{charge}(u, i) \leftarrow \text{charge}(u, i) - 1 \);
5. **end**
6. **foreach** cone \( i \) of vertex \( u \in DT \) where \( i \in \{0, 1, \ldots, 5\} \) such that \( \text{charge}(u, i) = 2 \wedge \text{charge}(u, i - 1) = 1 \wedge \text{charge}(u, i + 1) = 1 \) **do**
    7. **if** \( u = \text{last}(\text{parent}(u, i), i) \) **then**
       8. \( v_{\text{remove}} \leftarrow \text{last}(u, i - 1) \);
    9. **else**
       10. \( v_{\text{remove}} \leftarrow \text{first}(u, i + 1) \);
    **end**
    12. Remove edge \( \{u, v_{\text{remove}}\} \) from \( E \);
    13. \( \text{charge}(u, i) \leftarrow \text{charge}(u, i) - 1 \);
5. **end**
6. **return** \( E \);
ALGORITHM 10: $i$-relevant($v, u, i$)
1 $w \leftarrow \text{parent}(u, i)$;
2 return $v \neq \text{closest}(u, i) \land v \in C^i_w$;

ALGORITHM 11: $i$-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 12: parent($u, i \in \{0, 2, 4\}$)
1 return $\text{closest}(u, i)$;

ALGORITHM 13: closest($u, i$)
1 return the closest vertex to $u$ in cone $i$ of $u$, if it exists;

ALGORITHM 14: first($u, i$)
1 return the first vertex (considered clockwise) in cone $i$ of $u$, if it exists;

ALGORITHM 15: last($u, i$)
1 return the last vertex (considered clockwise) in cone $i$, if it exists;

ALGORITHM 16: KPX10($P$, integer $k \geq 14$)
1 $DT \leftarrow L_2$-DelaunayTriangulation($P$);
2 foreach vertex $u \in DT$ 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 maximal sequence of $\ell \geq 1$ consecutive empty cones 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 return the $DT$ edges selected by both endpoints;

• 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
The stretch-factors of the generated spanners are shown to be at most $1.998\left(\frac{2 \sin(\frac{2\pi}{5}) \cos(\frac{\pi}{5})}{2 \sin(2\pi/5) \cos(\pi/5) - 1}\right) \approx 5.7$. Refer to Algorithm 17.

**ALGORITHM 17: KX12(P)**

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

**ALGORITHM 18: 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 non-decreasing length do
   5. if $\forall C^i_u$ containing $\{u, v\}$, $C^i_u \cap E = \emptyset$ and $\forall C^j_v$ containing $\{u, v\}$, $C^j_v \cap E = \emptyset$ then
      6. Add edge $\{u, v\}$ to $E$;
   7. end
5. end
6. foreach $\{u, v\} \in E$ do
   7. $\text{Wedge}_\Delta(u, v)$;
   8. $\text{Wedge}_\Delta(v, u)$;
9. end
10. return $E \cup E^*$;

**BCC12–7, BCC12–6: Bose et al. [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. However, BCC12–6 constructs $11.998\left(\frac{1 - \tan(\frac{\pi}{7})}{\tan(\frac{\pi}{7}) + 1} \cos(\frac{\pi}{14})\right) \approx 81.7$-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 18. The parameter $\Delta \in \{7, 6\}$ is used to control the degree. Depending on $\Delta$, either Algorithm 20 or Algorithm 19 is invoked.
ALGORITHM 19: Wedge$_2(u, v_i)$

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

end

ALGORITHM 20: Wedge$_3(u, v_i)$

1. foreach $C^3_u$ containing $\{u, v_i\}$ do
2.   Let $Q = \{v_n : (u, v_n) \in C^3_u \cap DT\} = \{v_j, \ldots, v_k\}$;
3.   Let $Q' = \{v_n : \angle v_{n-1}v_nv_{n+1} < 6\pi/7, v_n \in Q \setminus \{v_j, v_i, v_k\}\}$;
4.   Add all edges $\{v_n, v_{n+1}\}$ to $E^*$ such that $v_n, v_{n+1} \notin Q'$ and $n \in [j + 1, i - 2] \cup [i + 1, k - 2]$; /* W.l.o.g. the points of $Q'$ lie between $v_i$ and $v_k$ (the symmetric case is handled analogously) */
5.   if $\angle uv_i v_{i-1} > 4\pi/7$ and $i, i - 1 \neq j$ then
6.     Add edge $\{v_i, v_{i-1}\}$ to $E^*$;
7. end
8. Let $v_f$ be the first point in $Q'$;
9. Let $a = \min\{n | n > f \text{ and } v_n \in Q \setminus Q'\}$;
10. if $f = i + 1$ then
11.    if $\angle uv_i v_{i+1} \leq 4\pi/7$ and $a \neq k$ then
12.       Add edge $\{v_f, v_a\}$ to $E^*$;
13.    end
14.    if $\angle uv_i v_{i+1} > 4\pi/7$ and $f + 1 \neq k$ then
15.       Add edge $\{v_i, v_{f+1}\}$ to $E^*$;
16.    end
17. else
18.    Let $v_\ell$ be the last point in $Q'$;
19.    Let $b = \max\{n | n < \ell \text{ and } v_n \in Q \setminus Q'\}$;
20.    if $\ell = k - 1$ then
21.       Add edge $\{v_\ell, v_b\}$ to $E^*$;
22.    else
23.       Add edge $\{v_b, v_\ell+1\}$ to $E^*$;
24.       if $v_{\ell-1} \in Q'$ then
25.          Add edge $\{v_\ell, v_{\ell-1}\}$ to $E^*$;
26.       end
27.    end
28. end

end
BKPX15: Bonichon et al. [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 defined as \( d_\infty (u, w) = \max(d_x(u, w), d_y(u, w)) \).

From the \( L_\infty \)-Delaunay triangulation of \( P \), a directed \( L_\infty \)-distance-based Yao graph \( Y^\infty_4 \) is constructed: the space around every point \( p \in P \) is divided into four disjoint 90\(^\circ\) cones and then for each non-empty cone, and a directed edge going out of \( p \) is placed between \( p \) and its closest neighbor in the cone according to the \( L_\infty \)-distance, breaking ties arbitrarily. The authors show that \( Y^\infty_4 \) is a plane \( \sqrt{20 + 14\sqrt{2}} \)-spanner. Then a degree-8 subgraph \( H_8 \) of \( Y^\infty_4 \) 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 21.

ALGORITHM 21: BKPX15(P)

1. \( DT \leftarrow L_\infty \)-DelaunayTriangulation(P);
2. \( Y^\infty_4 \leftarrow \text{constructYaoInfinityGraph}(DT) \) (Algorithm 22);
3. \( A \leftarrow \text{selectAnchors}(Y^\infty_4, DT) \) (Algorithm 23);
4. \( H_8 \leftarrow \text{degree8Spanner}(A, Y^\infty_4, DT) \) (Algorithm 26);
5. \( H_6 \leftarrow \text{processDupEdgeChains}(H_8, Y^\infty_4) \) (Algorithm 27);
6. \( H_4 \leftarrow \text{createShortcuts}(H_6, Y^\infty_4, DT) \) (Algorithm 28);
7. \( \text{return } H_4 \);

ALGORITHM 22: constructYaoInfinityGraph(DT)

1. \( Y^\infty_4 \leftarrow 0; \)
2. \( \text{foreach } u \in DT \text{ do} \)
3. \( \quad \text{foreach cone } C^i_u \text{ around } u \text{ do} \)
4. \( \quad \quad \text{Let } v \in C^i_u \text{ be the vertex with the smallest } L_\infty \text{ distance;} \)
5. \( \quad \quad \text{Add } (u, v) \text{ to } Y^\infty_4; \)
6. \( \quad \text{end} \)
7. \( \text{end} \)
8. \( \text{return } Y^\infty_4 \);
ALGORITHM 23: selectAnchors($Y_4^{\infty}, DT$)

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

ALGORITHM 24: fan(DT, u, i)

1. \( \text{return all neighboring vertices } (v_1, v_2, \ldots, v_k) \text{ in } C^i_u \text{ in counterclockwise order;} \)

ALGORITHM 25: isMutuallySingle($Y_4^{\infty}$, u, v, i)

1. \( \text{return } u \text{ has one edge from } Y_4^{\infty} \text{ in } C^i_u \text{ and } v \text{ has one edge from } Y_4^{\infty} \text{ in } C^{i+2}_u; \)
ALGORITHM 26: degree8Spanner \((A, Y_4^{\infty}, DT)\)

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 vertex \(u\) and cone \(i\) of \(u\) do
   4. \(\{v_1, \ldots, v_k\} \leftarrow \text{fan}(DT, u, i);\)
   5. if \(k \geq 2\) then
      6. Add all uni-directional canonical edges to \(H_8\) except \((v_2, v_1)\) and \((v_{k-1}, v_k)\);
      7. if \((v_2, v_1)\) is a non-anchor, uni-directional edge such that
         \((v_2, v_1) \in Y_4^{\infty} \land (v_1, v_2) \notin Y_4^{\infty} \land (v_1, u)\) is a dual edge \& not a start-of-odd-chain anchor chosen by \(v_1\) then
            8. Add \((v_2, v_1)\) to \(H_8;\)
      8. if \((v_{k-1}, v_k)\) is a non-anchor, uni-directional edge such that
         \((v_{k-1}, v_k) \in Y_4^{\infty} \land (v_k, v_{k-1}) \notin Y_4^{\infty} \land (v_k, u)\) is a dual edge \& not a start-of-odd-chain anchor chosen by \(v_k\) then
            9. Add \((v_{k-1}, v_k)\) to \(H_8;\)
   9. end
10. foreach canonical edge \((v, w)\) added to \(H_8\) do
    11. \(v_{\text{charge}} \leftarrow v;\)
    12. if \((v, w)\) is a non-anchor then
        13. \(v_{\text{charge}} \leftarrow u;\)
    14. end
    15. Charge \((v, w)\) to the cone of \(v\) containing \(w\) and the cone of \(w\) containing \(v_{\text{charge}};\)
   16. end
17. end
18. return \(H_8;\)

ALGORITHM 27: processDupEdgeChains\((H_8, Y_4^{\infty})\)

1. \(H_6 \leftarrow H_8;\)
2. foreach uni-directional non-anchor \((u, v)\) in cone \(i\) of \(u\) in \(H_8\) with charge \(= 1\) do
   3. if cone \(i + 1\) or \(i - 1\) of \(v\) has charge \(= 2\) \& \((u, v)\) is charged to cone \(i + 1\) or \(i - 1\) of \(v\) then
       4. Let \(j\) be the cone of \(v\) where \((u, v)\) is charged;
       5. \(v_{\text{current}} \leftarrow u, v_{\text{next}} \leftarrow v, D \leftarrow \emptyset;\)
       6. while \(j\) of \(v_{\text{next}}\) has charge \(= 2\) \& \((v_{\text{current}}, v_{\text{next}})\) is in cone \(j\) of \(v_{\text{next}}\) do
           7. Add \((v_{\text{current}}, v_{\text{next}})\) to \(D;\)
           8. \(v_{\text{current}} \leftarrow v_{\text{next}};\)
           9. Set \(v_{\text{next}}\) to the target of the \(Y_4^{\infty}\) edge beginning in cone \(j\) of \(v_{\text{current}};\)
           10. swap(i, j);
       11. end
   12. Starting with the last edge in the path induced by \(D\), remove every other edge from \(H_6;\)
   13. end
14. return \(H_6;\)
ALGORITHM 28: createShortcuts($H_6, Y_{\infty}^{40}, DT$)

1 $H_4 \leftarrow H_6$;
2 foreach pair of non-anchor uni-directional canonical edges $(v_{r-1}, v_r), (v_{r+1}, v_r)$ in cone $i$ of $u$ 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 $H_4$;

• KPT17: Kanj et al. [32]: Akin to BGHP10, this algorithm uses the $TD$-Delaunay triangulation and $\Theta$-graph to introduce fresh techniques in spanner construction. Refer to Algorithm 29 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.

ALGORITHM 29: KPT17($P$)

1 Notations. For each vertex, the shortest edge in each odd cone is called an anchor. Cones 1 and 4 are labeled 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.
2 $DT \leftarrow TD$-DelaunayTriangulation($P$);
3 $E, A \leftarrow \emptyset$;
4 foreach white anchor $(u, v)$ in increasing order of $d_\Omega$ length do
5   if $u$ and $v$ do not have a white anchor in a cone adjacent to $(u, v)$’s cone then
6     Add $(u, v)$ to $A$;
7 end
8 end
9 Add all blue anchors to $A$;
10 foreach blue anchor $u$ do
11   Let $s_1, s_2, \ldots, s_m$ be the clockwise ordered neighbors of $u$ in $DT$;
12   Add all canonical edges $(s_\ell, s_{\ell+1}) \notin A$ to $E$ such that $1 \leq \ell < m$;
13 end
14 foreach pair of canonical edges $(u, v), (w, v) \in E$ in a blue cone do
15   Remove $(u, v)$ and $(w, v)$ from $E$;
16   Add a shortcut edge $(u, w)$ to $E$;
17 end
18 foreach white canonical edge $(u, v)$ on the white side of its anchor $a$ do
19   if $a \notin A$ then
20     Add $(u, v)$ to $E$;
21 end
22 end
foreach white anchor \((v, w)\) and its boundary edge \((u, w) \neq (v, w)\) on the white side do

Let \(u = s_1, s_2, \ldots, s_m = v\) be the canonical path between \(u\) and \(v\);

for \(i = 0\) to \(m\) do

if \((s_{i+1}, s_i)\) is blue then

Let \(j\) be the smallest index in \(P_i = \{s_{i+1}, \ldots, s_m\}\) 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 \(s_is_j\);

Add the shortcut \((s_j, s_i)\) to \(E\);

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

Remove \((s_j, s_{j-1})\) from \(E\);

end

\(i \leftarrow j\);

end

end

return \(E \cup \{\}\; A\);

• **BHS18: Bose et al. [14]**. This algorithm produces a plane degree-8 spanner with stretch-factor at most \(1.998(1 + \frac{2\pi}{6\cos(\pi/6)}) \approx 4.4\) 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 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 30.

**ALGORITHM 30: BHS18(P)**

1 **Notations.** The bisector-distance \([pq]\) between \(p\) and \(q\) is the distance from \(p\) to the orthogonal projection of \(q\) onto the bisector of \(CP_i\) where \(q \in CP_i\). Let \(\{q_0, q_1, \ldots, q_{d-1}\}\) 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 \(\{p, q_0, q_1, \ldots, q_{d-1}\}\) 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. \(N^p_{CP_i}\) is the subgraph of \(N_p\) induced by all the vertices of \(N_p\) in \(CP_i\), including \(p\).

Let \(Can_{i}^{\{p, r\}}\) be the subgraph of \(DT\) consisting of the ordered subsequence of canonical edges \((s, t)\) of \(N^p_{CP_i}\) in clockwise order around apex \(p\) such that \(|ps| \geq |pr|\) and \(|pr| \geq |pr|\).

2 \(DT \leftarrow L_2\)-DelaunayTriangulation\((P)\);

3 Let \(m\) be the number of edges in \(DT\);

4 \(L\) be the edges \(\in DT\) sorted in non-decreasing order of bisector-distance;

5 \(E_A \leftarrow \text{addIncident}(L), E_{CAN} \leftarrow \{\}\;

6 foreach \([u, v] \in E_A\) do

7 \(E_{CAN} \leftarrow E_{CAN} \cup \text{addCanonical}(u, v) \cup \text{addCanonical}(v, u)\);

8 end

9 return \(E_A \cup E_{CAN}\);
ALGORITHM 31: addIncident(L)
1 $E_A \leftarrow \emptyset$
2 foreach $\{u, v\} \in L$ do
3     Let $i$ be the cone of $u$ containing $v$
4     if $\{u, w\} \not\in E_A$ for all $w \in N^u_i \land \{v, y\} \not\in E_A$ for all $y \in N^v_{i+3}$ then
5         Add $\{u, v\}$ to $E_A$
6     end
7 end
8 return $E_A$

ALGORITHM 32: 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 $Can_{i}^{\{u, v\}}$ has at least three edges then
5     foreach $\{s, t\} \in Can_{i}^{\{u, v\}} \setminus \{e_{first}, e_{last}\}$ do
6         Add $\{s, t\}$ to $E'$
7     end
8 end
9 if $v \in \{e_{first}, e_{last}\}$ 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 $\{y, z\} \in \{e_{first}, e_{last}\}$ do
13     if $\{y, z\} \in N^z_{i-2}$ then
14         Add $\{y, z\}$ to $E'$
15     end
16     if $\{y, z\} \in N^z_{i-2}$ then
17         if $N^z_{i-2} \cap E_A$ does not have an edge incident to $z$ then
18             Add $\{y, z\}$ to $E'$
19         end
20         if $N^z_{i-2} \cap E_A \setminus \{y, z\}$ 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 $E'$

3 ESTIMATING STRETCH-FACTORs OF LARGE SPANNERS
Measuring exact stretch-factors of large graphs is a tedious job, and also 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 $O(n^2 \log n + n|E|)$ 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 (see [1, 20, 28, 38, 44]). 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 ESTIMATE-STRETCHFACTOR, which 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 33, which takes as input an $n$-element pointset $P$ and a geometric graph $G$, constructed on $P$.

**Algorithm 33: ESTIMATE-STRETCHFACTOR(P, G)**

1. $DT \leftarrow L_2$-DelaunayTriangulation($P$);
2. $t \leftarrow 1$;
3. **foreach** $p \in P$ **do**
   1. $h \leftarrow 1$, $t_p \leftarrow 1$;
   2. **while** true **do**
      1. 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 $p$;
      2. $t' \leftarrow 1$;
      3. **foreach** $q \in X$ **do**
         1. $t' \leftarrow \max\left(\frac{|\pi_G(p, q)|}{|pq|}, t'\right)$;
      4. **end**
      5. **if** $t' > t_p$ **then**
         1. $h \leftarrow h + 1$; $t_p \leftarrow t'$;
      6. **else**
         1. **break**;
      7. **end**
   3. $t \leftarrow \max(t, t_p)$;
4. **end**
5. **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 found that for the class of spanners used in this work, ESTIMATESTRETCHFACTOR 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.

4 EXPERIMENTS

We have implemented the algorithms in GNU C++17 using the CGAL library \[41\]. The machine used for experiments is equipped with an 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. From CGAL, the Exact_predicates_inexact_constructions_kernel is used for accuracy and speed.

All 11 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, the CGAL::Delaunay_triangulation_2 and CGAL::Segment_Delaunay_graph_Linf_2 implementations have been used, respectively. As of now, a $TD$-Delaunay triangulation implementation is not available in the CGAL. It was pointed out by Chew \[22\] 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 the work of Bonichon et al. \[8\] that $TD$-Delaunay triangulation of a pointset is the same as its $\frac{1}{2} \cdot \Theta$ 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.

In our experiments, we have used both synthetic and real-world pointsets, as described next.

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 elsewhere \[4, 5, 29, 30, 40\] for geometric experiments. Figure 2 allows us to visualize these eight distributions:

1. uni-square: Points were generated uniformly inside a square of side length of 1,000 using the CGAL::Random_points_in_square_2 generator.
2. uni-disk: Points were generated uniformly inside a disc of radius 1,000 using the CGAL::Random_points_in_disc_2 generator.
3. 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 were set to 2.0). We have used std::normal_distribution<double> to generate the point coordinates.
4. normal: This is the same as normal-clustered except that only one cluster was used.
5. grid-contiguous: Points were generated contiguously on a $\lceil \sqrt{n} \rceil \times \lceil \sqrt{n} \rceil$ square grid using the CGAL::points_on_square_grid_2 generator.
6. grid-random: Points were generated on a $[0.7n] \times [0.7n]$ unit square grid. The value 0.7 was chosen arbitrarily to obtain well-separated non-contiguous grid points. The coordinates of the generated points are integers and were generated independently using std::uniform_int_distribution.
1.1:20 F. Anderson et al.

Fig. 2. The eight distributions used to generate synthetic pointsets for our experiments.

(7) annulus: Points were generated inside an annulus whose outer radius was set to 1,000 and the inner radius to 800. We have used std::uniform_real_distribution to generate the coordinates.

(8) galaxy: Points were generated in the shape of a spiral galaxy having outer five arms (see [31]).

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 were 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 were 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: 33,708-element pointset representing cities in Burma [29, 43].
- birch3: 99,801-element pointset representing random clusters at random locations [17, 30].
- monalisa: 100,000-city TSP instance representing a continuous-line drawing of the Mona Lisa [29, 30, 43].
- KDDCU2D: 104,297-element pointset representing the first two dimensions of a protein dataset [17, 29, 30].
- usa: 115,475-city TSP instance representing (nearly) all towns, villages, and cities in the United States [29, 30, 43].
- europe: 168,896-element pointset representing differential coordinates of the map of Europe [17, 29, 30].
- wiki: 317,695-element pointset of coordinates found in English language Wikipedia articles (source: https://github.com/placemarkt/wiki_coordinates).
- vlsi: 744,710-element pointset representing a very large-scale integration chip [43].
Bounded-Degree Plane Geometric Spanners in Practice

1.1:21

Fig. 3. The plot legends.

- **china**: 808,693-element pointset representing cities in China [17, 29, 30].
- **world**: 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 [29, 30, 43].
- **nyctaxi**: 2,728,717-element pointset representing Yellow Cab pickup locations in New York City in 2016 [29] (source: https://www.kaggle.com/c/nyc-taxi-trip-duration).

4.3 Efficacy of EstimateStretchFactor

We have seen in Section 3 that it is quite challenging to measure stretch-factor of large spanners. This motivated us to design and use the EstimateStretchFactor algorithm in our experiments for estimating stretch-factors of the generated spanners. In the following, we compare EstimateStretchFactor with Dijkstra’s algorithm (run from every vertex) and show that 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 EstimateStretchFactor 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 $11 \cdot 8 \cdot 10 \cdot 5 = 4,400$ since there are 11 algorithms, eight distributions, and 10 distinct values of $n$ ($1K, 2K, \ldots, 10K$), and five samples were used for every value of $n$. Out of these, the number of times EstimateStretchFactor has failed to return the exact stretch-factor is just 8. Thus, the observed failure rate is $\approx 0.18\%$. Interestingly, in the cases where EstimateStretchFactor 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 gave us the confidence that our algorithm can be safely used to estimate stretch-factor of large spanners. Refer to Figure 5. As is evident from these graphs, EstimateStretchFactor is substantially faster than Dijkstra everywhere. Henceforth, we use EstimateStretchFactor (Algorithm 33) to estimate the stretch-factors of the spanners in our experiments.

4.4 Experimental Comparison of the Algorithms

We compare the 11 implemented algorithms based on their runtime, degree, stretch-factor, and lightness of the generated spanners.

In the interest of space, we avoid legend tables everywhere in our plots. Since the legends are used uniformly everywhere, we present them here for an easy reference (Figure 3).

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

In our experiments, we found that BGHP10 and KPT17 were considerably slower than the other algorithms considered in this work. The reason behind this is slow construction of TD-Delaunay
Fig. 4. Points are generated using the uni-square distribution. Left: The running times for all the algorithms are shown; the plots for BGHP10 and KPT17 have overlapped in this figure, and they are the slowest ones in this case. Middle: Here, we consider the runtimes without the time taken to construct the respective Delaunay triangulation. Right: This is the same as the middle figure with $y$-axis scale adjusted for a better visual comparison.

triangulations. Figure 4 represents an illustration. When $n = 100K$, both took more than 150 seconds to finish. In contrast, the other nine algorithms took less than 10 seconds. Since real-world speed is an important factor for spanner construction algorithms, we do not consider them further in our runtime comparisons:

- **Runtime**: Fast execution speed is highly desired for spanner construction on large pointsets. We present the runtimes for all eight distributions in Figure 6. As explained earlier, we have excluded BGHP10 and KPT17 from these plots since they are considerably slower than the other nine algorithms. Interestingly, we found that the relative performance of these algorithms is independent of the point distributions. We further observed that not only are these algorithms slow because of the time taken to construct $TD$-Delaunay triangulation, but interestingly, their non-Delaunay steps are even slower than the other algorithms. Thus, this means that even if the construction of $TD$-Delaunay triangulation is engineered more efficiently, BGHP10 and KPT17 will still be the slowest in practice.

  For all eight distributions, we found that BKSPX15 was much slower than the others. This is mainly due to the time taken to construct $L_{\infty}$-Delaunay triangulation. Among the ones that use $L_2$-Delaunay triangulations, BG05 was the slowest due to the overhead of creation of temporary geometric graphs needed to control the degree and stretch-factor of the output 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 to the preceding 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. The table presented in Figure 9 presents the runtimes in seconds.

- **Degree**: Refer to Figure 7. In the tables, $\Delta$ denotes the theoretical degree upper bound, as claimed by the authors of these algorithms; $\Delta_{\text{observed}}$ denotes the maximum degree observed in our experiments; $\Delta_{\text{avg}}$ denotes the observed average degree; and $\Delta_{\text{vertex}}$ denotes the observed average degree per vertex. In our experiments, we found that spanners generated by BG05, LW04, and BSX09 have degrees much less than the degree upper bounds derived by the authors. Although 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...
Fig. 5. Runtime comparison: Dijkstra (run from every vertex) vs ESTIMATESTRETCHFACTOR. For every value of $n$, we have used $11 \cdot 5 = 55$ spanner samples since there are 11 algorithms and five pointsets were generated for that value of $n$ using the same distribution.
Fig. 6. Runtime comparisons of the nine algorithms (BGHP10 and KPT17 are excluded).
Fig. 7. Degree comparisons of the spanners generated by the 11 algorithms.

are 27, 23, and 17, respectively. For the remaining eight algorithms, the claimed degree upper bounds were achieved in our experiments, thereby showing that the analyses obtained by the authors of those algorithms are tight. However, the degree bound claimed by the authors of BCC12-6 appears to be incorrect. We present an example in the appendix (Section A.2)
where the degree of the spanner generated by this algorithm exceeds 6 (in fact, it is 7 in this example). For every algorithm, we found that the average degree of the generated spanners was not far away from the maximum observed degrees. It shows that the algorithms appear to spread the edges evenly in constructing the spanners. The average degree per vertex is another way to judge the quality of the spanners. In this regard, we found that it was always between 6 and 3 everywhere and is quite reasonable for practical purposes. This shows that

Fig. 8. Stretch-factor and lightness comparisons of the spanners generated by the 11 algorithms.

(a) uni-square  
(b) uni-disk  
(c) normal-clustered  
(d) normal  
(e) grid-contiguous  
(f) grid-random  
(g) annulus  
(h) galaxy
all these algorithms are very careful when it comes to the selection of spanner edges. The lowest values were achieved by BKpX15 and KPT17. For the real-world pointsets, we found 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 Figures 10 and 11 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.

- **Stretch-factor**: Refer to Figure 8. In the tables, \( t \) denotes the theoretical stretch-factor, as derived by the authors of these algorithms; \( t_{\text{max}} \) denotes the maximum stretch-factor observed in our experiments; and \( t_{\text{avg}} \) denotes the average observed stretch-factor. Among the 11 algorithms, KPXi0 has the lowest guaranteed stretch-factor—it is 2.9. The stretch-factors of the spanners generated by KPXi0 are always less than 1.6, thereby making it the best among the 11 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 KPT17 that can also generate degree-4 plane spanners has a lower observed maximum stretch-factor—it is 5.236 (the theoretical upper bound is 20 for this algorithm). Overall, we found 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 found 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 Figure 12. Once again, KPX10 produced the lowest stretch-factor spanners. The stretch-factors seem quite reasonable everywhere except in the two cases of vlsi and nyctaxi pointsets when fed to BKPX15. The produced spanners have stretch-factors of 11.535 and 20.009, respectively. The latter is interesting since the lower bound example constructed by Bonichon et al. [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 nyctaxi pointset beats this lower bound.

- **Lightness**: 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 $P$. 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 Figure 8. Lightness is denoted by $\ell$. With a few exceptions, we found 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 generated by BGS05 and LW04 is marginal (around 2), the difference between their lightness is substantial (approximately 6 for some cases). However, the degree-4 spanners generated by KPT17 have the lowest lightness (less than 2.9 everywhere). Interestingly, although BKPX15 generates degree-4 spanners, their lightness was 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 lightness of the spanners generated for real-world pointsets follows a similar trend, and we did not observe anything special. Figure 13 presents more details.
Remark. In our experiments, we found that the spanners’ degree, stretch-factor, and lightness remained somewhat constant with the increase in \( n \). Hence, we do not present plots for them.

5 CONCLUSION

Since there are various ways (speed, degree, stretch-factor, lightness) to judge the 11 algorithms, it is hard to declare the winner(s). Thus, based on our experimental observations, we come to the following conclusions (which 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 found 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 3\( n \) 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 are not a requirement, we recommend using BHS18 or BCC12-7 since they produced spanners of reasonable lightness (less than 4 most of the time).

6 CODE AND VISUALIZATIONS

For the C++ implementations, refer to our GitHub repository at https://github.com/ghoshanirban/BoundedDegreePlaneSpannersCppCode. Refer to the applet hosted at https://ghoshanirban.github.io/bounded-degree-plane-spanners/index.html for an in-browser visual experience.
A APPENDIX
A.1 Sample Outputs

Fig. A.1. A 150-element pointset, drawn randomly from a square.

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

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

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

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

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

Fig. A.8. The spanner generated by BCC12-7 on the pointset shown in Figure A.1; degree: 6, stretch-factor: 2.302473.
Fig. A.9. The spanner generated by BCC12-6 on the pointset shown in Figure A.1; degree: 6, stretch-factor: 1.735716.

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

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

Fig. A.12. The spanner generated by KPT17 on the pointset shown in Figure A.1; degree: 4, stretch-factor: 2.582846.
A.2 A Counterexample for BCC12−6

In the following, we present a 13-element pointset on which BCC12−6 fails to construct a degree-6 plane spanner. Figure A.13 presents the pointset.

First, BCC12−6 creates the $L_2$-Delaunay triangulation of $P$ and initializes seven cones around every $p_i$, oriented such that the shortest edge incident on $p_i$ falls on a boundary. See Figures A.14 and A.15.

Fig. A.14. The $L_2$-Delaunay triangulation of $P$.

Next, in Figure A.16, we show the edges added by the main portion of the algorithm (excluding the edges added by $\text{Wedge}_6$ calls). Only $\text{Wedge}_6(p_1, p_2)$ and $\text{Wedge}_6(p_{12}, p_{11})$ calls add new edges to $E^*$ and thus to the final spanner as well. The former call adds the two edges $p_3p_6, p_6p_{12}$ (Figure A.17), and the latter call adds the edge $p_6p_{10}$ (Figure A.18). The final spanner is shown in Figure A.19. Note that $p_6$ has degree 7 in the spanner, which violates the degree requirement of the spanners produced by BCC12−6.

ACM Journal of Experimental Algorithmics, Vol. 28, No. 1, Article 1.1. Publication date: April 2023.
Fig. A.16. Edges added by the main portion of BCC12 (excluding calls to subroutine \textit{Wedge}_6).

Fig. A.17. The edge $p_1 p_2$ (shown in red) is added during the main portion of the algorithm, and the call to \textit{Wedge}_6($p_1$, $p_2$) adds the two blue edges $p_3 p_6$ and $p_6 p_{12}$.

Fig. A.18. The edge $p_{12} p_{11}$ (shown in red) is added during the main portion of the algorithm, and the call to \textit{Wedge}_6($p_{12}$, $p_{11}$) adds the blue edge $p_6 p_{10}$.

Fig. A.19. 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 Figure A.16 along with the blue edges shown in Figures A.17 and A.18.

\textbf{ACKNOWLEDGMENTS}

We sincerely thank Nicolas Bonichon (one of the authors of BKPX15) for sharing the applet code for the algorithm BKPX15 [11]. The code has helped us understand the algorithm clearly and create a CGAL implementation of the algorithm. We are grateful to the three anonymous reviewers of our manuscript, whose suggestions have helped us improve this article’s presentation.
REFERENCES

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

[2] Fred Anderson, Anirban Ghosh, Matthew Graham, Lucas Mougeot, and David Wisnosky. 2021. An interactive tool for experimenting with bounded-degree plane geometric spanners (media exposition). In *Proceedings of the 37th International Symposium on Computational Geometry (SoCG’21)*.

[3] Davood Bakhshesh and Mohammad Farshi. 2021. A degree 3 plane 5.19-spanner for points in convex position. *Scientia Iranica* 28, 6 (2021), 3324–3331.

[4] Jon Louis Bentley. 1992. Fast algorithms for geometric traveling salesman problems. *ORSA Journal on Computing* 4, 4 (1992), 387–411.

[5] Jon Louis Bentley. 1990. K-d trees for semidynamic point sets. In *Proceedings of the 6th Annual Symposium on Computational Geometry*. 187–197.

[6] Ahmad Biniaz. 2020. Plane hop spanners for unit disk graphs: Simpler and better. *Computational Geometry* 89 (2020), 101622.

[7] Ahmad Biniaz, Prosenjit Bose, Jean-Lou De Carufel, Cyril Gavoille, Anil Maheshwari, and Michiel Smid. 2017. Towards plane spanners of degree six. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*. 19–30.

[8] Nicolas Bonichon, Cyril Gavoille, Nicolas Hanusse, and David Ilcinkas. 2010. Connections between theta-graphs, Delaunay triangulations, and orthogonal surfaces. In *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science*. 266–278.

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

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

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

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

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

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

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

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

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

[18] Paul B. Callahan and S. Rao Kosaraju. 1995. A decomposition of multidimensional point sets with applications to $k$-nearest-neighbors and $n$-body potential fields. *Journal of the ACM* 42, 1 (1995), 67–90.

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

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

[21] L. Paul Chew. 1986. There is a planar graph almost as good as the complete graph. In *Proceedings of the 2nd Annual Symposium on Computational Geometry*. 192–197.

[22] L. Paul Chew. 1989. There are planar graphs almost as good as the complete graph. In *Proceedings of the 2nd Annual Symposium on Computational Geometry*. 192–197.

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

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

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

[26] Adrian Dumitrescu, Anirban Ghosh, and Csaba D. Tóth. 2022. Sparse hop spanners for unit disk graphs. *Computational Geometry* 100 (2022), 101808.
[27] Mohammad Farshi and Joachim Gudmundsson. 2009. Experimental study of geometric $t$-spanners. *Journal of Experimental Algorithms* 14 (2009), 3.

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

[29] Rachel Friederich, Anirban Ghosh, Matthew Graham, Brian Hicks, and Ronald Shevchenko. 2023. Experiments with unit disk cover algorithms for covering massive point sets. *Computational Geometry* 109 (2023), 101925.

[30] Anirban Ghosh, Brian Hicks, and Ronald Shevchenko. 2019. Unit disk cover for massive point sets. In *Proceedings of the International Symposium on Experimental Algorithms*. 142–157.

[31] Itinerant Games. 2014. A 2D Procedural Galaxy with C++. Retrieved February 8, 2023 from [https://itinerantgames.tumblr.com/post/78592276402/a-2d-procedural-galaxy-with-c](https://itinerantgames.tumblr.com/post/78592276402/a-2d-procedural-galaxy-with-c).

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

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

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

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

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

[37] Wolfgang Mulzer. 2004. *Minimum Dilation Triangulations for the Regular $n$-Gon*. Master’s Thesis. Freie Universität Berlin, Germany.

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

[39] Giri Narasimhan and Michiel Smid. 2007. *Geometric Spanner Networks*. Cambridge University Press.

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

[41] The CGAL Project. 2021. *CGAL User and Reference Manual* (5.3 ed.). CGAL Editorial Board. [https://doc.cgal.org/5.3/Manual/packages.html](https://doc.cgal.org/5.3/Manual/packages.html).

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

[43] TSP. 2022. Traveling Salesman Problem. Retrieved December 8, 2022 from [https://www.math.uwaterloo.ca/tsp/](https://www.math.uwaterloo.ca/tsp/).

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

[45] Ge Xia. 2013. The stretch factor of the Delaunay triangulation is less than 1.998. *SIAM Journal on Computing* 42, 4 (2013), 1620–1659.

Received 5 May 2022; revised 7 November 2022; accepted 6 December 2022