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

Research on this work was supported by the University of North Florida Academic Technology Grant and NSF Award CCF-1947887.
Authors' address: F. Anderson, A. Ghosh, M. Graham, L. Mougeot, and D. Wisnosky, School of Computing, University of North Florida, 1 UNF Drive, Jacksonville, FL 32224; emails: {n01451351, anirban.ghosh, n00612546, n01398041, n01153911}@unf.edu.

1084-6654/2023/04-ART1.1 $15.00
https://doi.org/10.1145/3582497

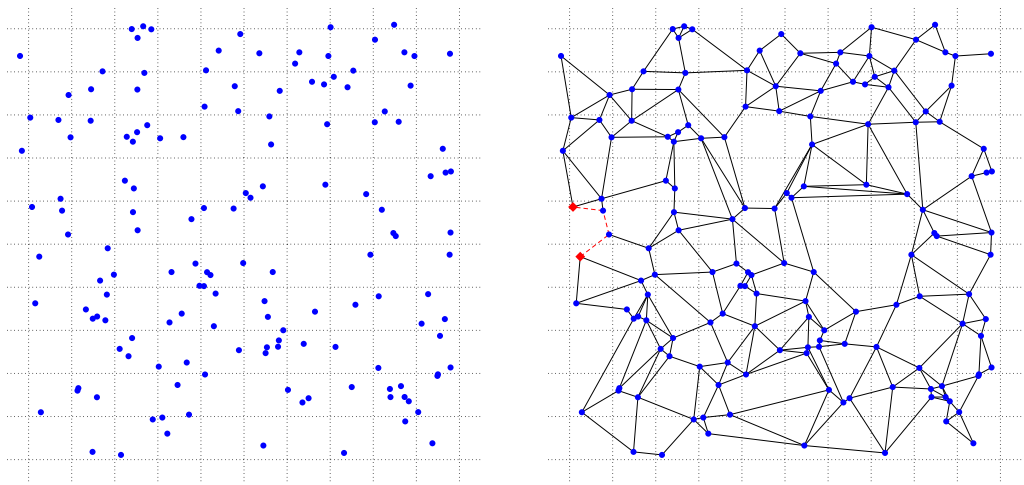

Fig. 1. Left: A set P of 150 points, generated randomly within a square. Right: A plane degree-6 spanner on P with stretch-factor ≈ 1.82. The pair of points for which the spanner achieves a stretch-factor of ≈ 1.82 is shown in red along with the shortest path between them.

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 σ 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

Table 1. Summary of Results on Constructions of Bounded-Degree Plane Geometric Spanners, Sorted by the Degree They Guarantee

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

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.

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 EstimateStretchFactor, 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

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_i^u. 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)

```
Declare an empty array Φ[1, ..., n];
Make a copy of DT and call it H;
Let reserved be a set of two consecutive vertices v1, v2 on the convex hull of H;
Φ[1] ← v1, Φ[2] ← v2;
for i = 1 to n − 2 do
    Let u be a vertex of the outer face of H \ reserved that is adjacent to at most two other
      vertices on the outer face;
    Φ[u] ← n − i + 1;
    Remove u and all incident edges from H;
end
return Φ;
```

ALGORITHM 2: SpanningGraph(DT)

$\Phi[1,\ldots,n] \leftarrow$ CanonicalOrdering(DT) (Algorithm 1);
$SG \leftarrow \emptyset$;
Add edges between $v_1, v_2, v_3 \in \Phi$ to SG and mark the vertices as done;
for $v_i \in \Phi \setminus \{v_1, v_2, v_3\}$ **do**
  Let $u_1, \ldots, u_k$ be the vertices neighboring v_i in DT marked as done;
  Remove edge $\{u_1, u_2\}$ from SG;
  Add edges $\{v_i, u_1\}$ and $\{v_i, u_2\}$ to SG;
  if $k > 2$ **then**
    Remove edge $\{u_{k-1}, u_k\}$ from SG;
    Add edge $\{v_i, u_k\}$ to SG;
  end
end
return SG;

ALGORITHM 3: TransformPolygon(SG, DT)

$V \leftarrow \emptyset, E \leftarrow \emptyset$;
Let s_1, v_1 be two consecutive vertices on the convex hull of SG in counterclockwise order;
$v_{prev} \leftarrow s_1, v_i \leftarrow v_1$;
Add v_{prev} to V;
do
  Add v_i to V;
  Add $\{v_i, v_{prev}\}$ to E;
  Let v_{next} be the neighbor of $v_i \in SG$ such that v_{next} is the next neighbor clockwise from v_{prev};
  $v_{prev} \leftarrow v_i, v_i \leftarrow v_{next}$;
while $v_{prev} \neq s_1$ **and** $v_i \neq v_1$;
$E = E \cup \{\{v_i, v_{prev}\}\} \cup DT \setminus SG$;
return (V, E);

ALGORITHM 4: PolygonSpanner(Q, SG)

Let V, E be the vertices and edges of Q, respectively;
Let $\rho[1,\ldots,n]$ be the breadth-first ordering of V in Q, starting at any vertex in V;
$E' \leftarrow SG$;
foreach $u \in \rho$ **do**
  Let $s_1, s_2, \ldots, s_m$ be the clockwise ordered neighbors of u in Q;
  $s_j, s_k \leftarrow s_m$;
  if $u \neq \rho_1$ **then**
    Set s_j and s_k to the first and last vertex in the ordered neighborhood of u where $s_j, s_k \in E'$;
  end
  Divide $\angle s_1 u s_j$ and $\angle s_k u s_m$ into an minimum number of cones with maximum angle $\pi/2$;
  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** $k \leq \ell < m$;
end
return E';

ALGORITHM 5: BGS05(P)

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

- 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 α 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.

ALGORITHM 6: ReverseLowDegreeOrdering(DT)

Declare an empty array $\Phi[1 \ldots n]$;
Make a copy of DT and call it H;
for $i = 1$ **to** n **do**
    Let u be a vertex in H with minimal degree;
    $\Phi[u] \leftarrow n - i + 1$;
    Remove u and all incident edges from H;
return Φ;

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

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

- 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 α 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 neighbors are unprocessed.

ALGORITHM 8: `BSX09`$(P,\ 0 < \alpha \leq 2\pi/3)$

```
1  DT ← L2-DelaunayTriangulation(P);
2  Φ[1...n] ← ReverseLowDegreeOrdering(DT) (Algorithm 6);
3  E ← ∅;
4  foreach u ∈ Φ do
5      if u 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 ⌊2π/α⌋ non-overlapping cones C_u^0, C_u^1, ... such that
             v_closest is on the boundary between the first and last cones;
9          foreach C_u^i except the first and last do
10             if u 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) mod m}} to E such that 0 ≤ j < m and v_j, v_{(j+1) mod m} are
                 unprocessed;
16         end
17     end
18     Mark u as processed;
19 end
20 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)

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.
$DT \leftarrow$ TD-DelaunayTriangulation(P);
$E \leftarrow \emptyset$;
foreach nonempty cone i of vertex $u \in DT$ where $i \in \{1, 3, 5\}$ **do**
    Add edge $\{u,$ closest$(u, i)\}$ to E;
    charge$(u, i) \leftarrow$ charge$(u, i) + 1$;
    charge(closest$(u, i), i + 3)) \leftarrow$ charge(closest$(u, i), i + 3) + 1$;
    if first$(u, i) \neq$ closest$(u, i) \wedge$ i-relevant(first$(u, i), u, i - 1)$ **then**
        Add edge $u,$ first(u, i) to E;
        charge$(u, i - 1) \leftarrow$ charge$(u, i - 1) + 1$;
    end
    if last$(u, i) \neq$ closest$(u, i) \wedge$ i-relevant(last$(u, i), u, i + 1)$ **then**
        Add edge $\{u,$ last$(u, i)\}$ to E;
        charge$(u, i + 1) \leftarrow$ charge$(u, i + 1) + 1$;
    end
end
foreach cone i of vertex $u \in DT$ where $i \in \{0, 2, 4\}$ such that i-distant(u, i) is true **do**
    $v_{next} \leftarrow$ first$(u, i + 1)$;
    $v_{prev} \leftarrow$ last$(u, i - 1)$;
    Add edge $\{v_{next}, v_{prev}\}$ to E;
    charge$(v_{next}, i + 1) \leftarrow$ charge$(v_{next}, i + 1) + 1$;
    charge$(v_{prev}, i - 1) \leftarrow$ charge$(v_{prev}, i - 1) + 1$;
    Let v_{remove} be the vertex from v_{next}, v_{prev} where $\angle$(parent$(u, i), u, v_{remove}$) is maximized;
    Remove edge $\{u, v_{remove}\}$ from E;
    charge$(u, i) \leftarrow$ charge$(u, i) - 1$;
end
foreach cone i of vertex $u \in DT$ where $i \in \{0, 1, \ldots, 5\}$ such that charge$(u, i) = 2 \wedge$ charge$(u, i - 1) = 1 \wedge$ charge$(u, i + 1) = 1$ **do**
    if $u =$ last(parent$(u, i), i)$ **then**
        $v_{remove} \leftarrow$ last$(u, i - 1)$;
    else
        $v_{remove} \leftarrow$ first$(u, i + 1)$;
    end
    Remove edge $\{u, v_{remove}\}$ from E;
    charge$(u, i) \leftarrow$ charge$(u, i) - 1$;
end
return E;

ALGORITHM 10: `i-relevant`(v, u, i)

$w \leftarrow$ `parent`(u, i);
return $v \neq$ `closest`$(u, i) \wedge v \in C_w^i$;

ALGORITHM 11: `i-distant`(w, i)

$u \leftarrow$ `parent`(w, i);
return $\{w, u\} \notin E \wedge$ `i-relevant`$($`first`$(w, i+1), u, i+1) \wedge$ `i-relevant`$($`last`$(w, i-1), u, i-1)$;

ALGORITHM 12: `parent`$(u, i \in \{0, 2, 4\})$

return `closest`(u, i)

ALGORITHM 13: `closest`(u, i)

return the closest vertex to u in cone i of u, if it exists;

ALGORITHM 14: `first`(u, i)

return the first vertex (considered clockwise) in cone i of u, if it exists;

ALGORITHM 15: `last`(u, i)

return the last vertex (considered clockwise) in cone i, if it exists;

ALGORITHM 16: `KPX10`$(P$, integer $k \geq 14)$

```
DT ← L2-DelaunayTriangulation(P);
foreach vertex u ∈ DT do
    Partition the area surrounding u into k disjoint cones of angle 2π/k;
    In each nonempty cone, select the shortest edge in DT incident to u;
    foreach maximal sequence of ℓ ≥ 1 consecutive empty cones do
        if ℓ > 1 then
            select the first ⌊ℓ/2⌋ unselected incident DT edges on u clockwise from the
              sequence of empty cones and the first ⌈ℓ/2⌉ unselected DT edges incident on u
              counterclockwise from the sequence of empty cones;
        else
            let ux and uy be the incident DT edges on m clockwise and counterclockwise,
              respectively, from the empty cone;
            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;
        end
    end
end
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

subgraph is at most 11. The stretch-factors of the generated spanners are shown to be at most $1.998(\frac{2\sin(2\pi/5)\cos(\pi/5)}{2\sin(2\pi/5)\cos(\pi/5)-1}) \approx 5.7$. Refer to Algorithm 17.

ALGORITHM 17: KX12(P)

```
DT ← L2-DelaunayTriangulation(P);
foreach vertex u ∈ DT do
    In each wide sequence (a sequence of exactly three consecutive edges incident to a vertex
      whose overall angle is at least 4π/5) around u, select the edges of the sequence;
    Partition the remaining space surrounding u not in a wide sequence into a minimum
      number of disjoint cones of maximum angle π/5;
    In each nonempty cone, select the shortest edge incident to u;
    In each empty cone, let ux and uy be the incident DT edges on u clockwise and
      counterclockwise, respectively, from the empty cone;
    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;
end
return all edges selected by both incident vertices;
```

- 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 $11.998(\frac{1}{1-\tan(\pi/7)(1+1/\cos(\pi/14))}) \approx 81.7$-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 Δ, either Algorithm 20 or Algorithm 19 is invoked.

ALGORITHM 18: BCC12($P, \Delta \in \{6, 7\}$)

```
DT ← L2-DelaunayTriangulation(P);
E, E* ← ∅;
Initialize k = Δ + 1 cones surrounding each vertex u, oriented such that the shortest edge
  incident on u falls on a boundary;
foreach {u, v} ∈ DT in order of non-decreasing length do
    if ∀C_u^i containing {u, v}, C_u^i ∩ E = ∅ and ∀C_v^j containing {u, v}, C_v^j ∩ E = ∅ then
        Add edge {u, v} to E;
    end
end
foreach {u, v} ∈ E do
    Wedge_Δ(u, v);
    Wedge_Δ(v, u);
end
return E ∪ E*;
```

ALGORITHM 19: $\texttt{Wedge}_7(u, v_i)$

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

ALGORITHM 20: $\texttt{Wedge}_6(u, v_i)$

foreach C_u^z containing $\{u, v_i\}$ **do**
  Let $Q = \{v_n : \{u, v_n\} \in C_u^z \cap DT\} = \{v_j, \ldots, v_k\}$;
  Let $Q' = \{v_n : \angle v_{n-1} v_n v_{n+1} < 6\pi/7, v_n \in Q \setminus \{v_j, v_i, v_k\}\}$;
  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) */
  if $\angle u v_i v_{i-1} > 4\pi/7$ and $i, i-1 \neq j$ **then**
    Add edge $\{v_i, v_{i-1}\}$ to E^*;
  end
  Let v_f be the first point in Q';
  Let $a = \min\{n | n > f$ and $v_n \in Q \setminus Q'\}$;
  if $f = i+1$ **then**
    if $\angle u v_i v_{i+1} \leq 4\pi/7$ and $a \neq k$ **then**
      Add edge $\{v_f, v_a\}$ to E^*;
    end
    if $\angle u v_i v_{i+1} > 4\pi/7$ and $f+1 \neq k$ **then**
      Add edge $\{v_i, v_{f+1}\}$ to E^*;
    end
  else
    Let v_ℓ be the last point in Q';
    Let $b = \max\{n | n < \ell$ and $v_n \in Q \setminus Q'\}$;
    if $\ell = k-1$ **then**
      Add edge $\{v_\ell, v_b\}$ to E^*;
    else
      Add edge $\{v_b, v_{\ell+1}\}$ to E^*;
      if $v_{\ell-1} \in Q'$ **then**
        Add edge $\{v_\ell, v_{\ell-1}\}$ to E^*;
      end
    end
  end
end

- BKPX15: *Bonichon et al.* [11]: This algorithm uses the L_∞-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_∞-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_∞-Delaunay triangulation of P, a directed L_∞-distance-based Yao graph $\overrightarrow{Y_4^\infty}$ is constructed: the space around every point $p \in P$ is divided into four disjoint 90° 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_∞-distance, breaking ties arbitrarily. The authors show that $\overrightarrow{Y_4^\infty}$ is a plane $\sqrt{20 + 14\sqrt{2}}$-spanner. Then a degree-8 subgraph H_8 of Y_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)

Notations. The algorithm divides the space around each point into four 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 $w_0, w_1, w_2, \ldots, w_k$ of maximal length consisting of `weak anchors` such that the cone of each edge (w.r.t. 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.

$DT \leftarrow L_\infty$-`DelaunayTriangulation`(P);

$\overrightarrow{Y_4^\infty} \leftarrow$ `constructYaoInfinityGraph`(DT) (Algorithm 22);

$A \leftarrow$ `selectAnchors`($\overrightarrow{Y_4^\infty}, DT$) (Algorithm 23);

$H_8 \leftarrow$ `degree8Spanner`($A, \overrightarrow{Y_4^\infty}, DT$) (Algorithm 26);

$H_6 \leftarrow$ `processDupEdgeChains`($H_8, \overrightarrow{Y_4^\infty}$) (Algorithm 27);

$H_4 \leftarrow$ `createShortcuts`($H_6, \overrightarrow{Y_4^\infty}, DT$) (Algorithm 28);

return H_4;

ALGORITHM 22: `constructYaoInfinityGraph`(DT)

$\overrightarrow{Y_4^\infty} \leftarrow \emptyset$;
foreach $u \in DT$ **do**
  foreach cone C_u^i around u **do**
    Let $v \in C_u^i$ be the vertex with the smallest L_∞ distance;
    Add (u, v) to $\overrightarrow{Y_4^\infty}$;
  end
end
return $\overrightarrow{Y_4^\infty}$;

ALGORITHM 23: selectAnchors($\overrightarrow{Y_4^\infty}$, DT)

foreach $(u, v) \in \overrightarrow{Y_4^\infty}$ **do**
  Let i be the cone of u containing v;
  $v_{anchor} \leftarrow v$;
  if $\neg$isMutuallySingle($\overrightarrow{Y_4^\infty}, u, v, i$) **and** u has more than one $\overrightarrow{Y_4^\infty}$ edge in C_u^i **then**
    Let ℓ be the position of v and k the number of vertices in fan(DT, u, i);
    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**
      Let $v_{\ell'}$ such that $\ell' < \ell$ be the starting vertex of the maximal uni-directional canonical path ending at v_ℓ;
      $v_{anchor} \leftarrow v_{\ell'}$;
    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**
      Let $v_{\ell'}$ such that $\ell' > \ell$ be the starting vertex of the maximal uni-directional canonical path ending at v_ℓ;
      $v_{anchor} \leftarrow v_{\ell'}$;
    end
  Mark (u, v_{anchor}) as the anchor of C_u^i;
end
$A \leftarrow \emptyset$;
foreach anchor (u, v) in each C_u^i **do**
  if anchor of C_v^{i+2} is (v, u) **or** undefined **then**
    Mark (u, v) as strong and add it to A;
  else
    Mark (u, v) as weak;
  end
end
foreach weak anchor (u, v) in each C_u^i **do**
  if u begins the weak anchor chain $(w_0, w_1, \ldots, w_k)$ **then**
    if k is odd **then**
      Mark (w_0, w_1) as a start-of-odd-chain-anchor;
    end
    for $\ell = k - 1;\ \ell \geq 0;\ \ell = \ell - 2$ **do**
      Add $(w_{\ell-1}, w_\ell)$ to A;
    end
  end
end
return A;

ALGORITHM 24: fan(DT, u, i)

return all neighboring vertices $(v_1, v_2, \ldots, v_k)$ in C_u^i in counterclockwise order;

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

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

ALGORITHM 26: degree8Spanner($A, \overrightarrow{Y_4^\infty}, DT$)

```
1  Charge each anchor (u, v) ∈ A to the cones of each vertex in which the edge lies;
2  H_8 ← A;
3  foreach vertex u and cone i of u do
4      {v_1, ..., v_k} ← fan(DT, u, i);
5      if k ≥ 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) ∈ Y_4^∞ ∧ (v_1, v_2) ∉ Y_4^∞ ∧ (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;
9          end
10         if (v_{k-1}, v_k) is a non-anchor, uni-directional edge such that
             (v_{k-1}, v_k) ∈ Y_4^∞ ∧ (v_k, v_{k-1}) ∉ Y_4^∞ ∧ (v_k, u) is a dual edge ∧ not a
             start-of-odd-chain anchor chosen by v_k then
11             Add (v_{k-1}, v_k) to H_8;
12         end
13         foreach canonical edge (v, w) added to H_8 do
14             v_charge ← v;
15             if (v, w) is a non-anchor then
16                 v_charge ← u;
17             end
18             Charge (v, w) to the cone of v containing w and the cone of w containing v_charge;
19         end
20     end
21 end
22 return H_8;
```

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

```
1  H_6 ← 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_current ← u, v_next ← v, D ← ∅;
6          while cone j of v_next has charge = 2 ∧ (v_current, v_next) is in cone j of v_next do
7              Add (v_current, v_next) to D;
8              v_current ← v_next;
9              Set v_next to the target of the Y_4^∞ edge beginning in cone j of v_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 end
15 return H_6;
```

ALGORITHM 28: `createShortcuts`$(H_6, \overrightarrow{Y_4^{\infty}}, DT)$

```
H_4 ← H_6;
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
    Remove (v_{r−1}, v_r) and (v_{r+1}, v_r) from H_4;
    Add (v_{r−1}, v_{r+1}) to H_4;
    Charge this edge to the cones of each vertex in which the edge lies;
end
return H_4;
```

- KPT17: *Kanj et al.* [32]: Akin to BGHP10, this algorithm uses the TD-Delaunay triangulation and Θ-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)

```
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.
DT ← TD-DelaunayTriangulation(P);
E, A ← ∅;
foreach white anchor (u, v) in increasing order of d_▽ length do
    if u and v do not have a white anchor in a cone adjacent to (u, v)'s cone then
        Add (u, v) to A;
    end
end
Add all blue anchors to A;
foreach blue anchor u do
    Let s_1, s_2, ..., s_m be the clockwise ordered neighbors of u in DT;
    Add all canonical edges (s_ℓ, s_{ℓ+1}) ∉ A to E such that 1 ≤ ℓ < m;
end
foreach pair of canonical edges (u, v), (w, v) ∈ E in a blue cone do
    Remove (u, v) and (w, v) from E;
    Add a shortcut edge (u, w) to E;
end
foreach white canonical edge (u, v) on the white side of its anchor a do
    if a ∉ A then
        Add (u, v) to E;
    end
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 $s_i s_j$;
      Add the shortcut $\left(s_j, s_i\right)$ to E;
      if $\left(s_j, s_{j-1}\right) \in E$ **then**
        Remove $\left(s_j, s_{j-1}\right)$ from E;
      end
      $i \leftarrow j$;
    end
  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 Θ-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)

Notations. 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 $\{q_0, q_1, \ldots, q_{d-1}\}$ 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_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 $[pt] \geq [pr]$.
$DT \leftarrow L_2$-DelaunayTriangulation(P);
Let m be the number of edges in DT;
L be the edges $\in DT$ sorted in non-decreasing order of bisector-distance;
$E_A \leftarrow$ addIncident(L), $E_{CAN} \leftarrow \emptyset$;
foreach $\{u, v\} \in E_A$ **do**
  $E_{CAN} \leftarrow E_{CAN} \cup$ addCanonical$(u, v) \cup$ addCanonical(v, u);
end
return $E_A \cup E_{CAN}$;

ALGORITHM 31: addIncident(L)

```
E_A ← ∅;
foreach {u, v} ∈ L do
    Let i be the cone of u containing v;
    if {u, w} ∉ E_A for all w ∈ N_i^u ∧ {v, y} ∉ E_A for all y ∈ N_{i+3}^v then
        Add {u, v} to E_A;
    end
end
return E_A;
```

ALGORITHM 32: addCanonical(u, v)

```
E' ← ∅;
Let i be the cone of u containing v;
Let e_first and e_last be the first and last canonical edge in Can_i^{u,v};
if Can_i^{u,v} has at least three edges then
    foreach {s, t} ∈ Can_i^{u,v} \ {e_first, e_last} do
        Add {s, t} to E';
    end
end
if v ∈ {e_first, e_last} and there is more than one edge in Can_i^{u,v} then
    Add the edge of Can_i^{u,v} incident to v to E';
end
foreach {y, z} ∈ {e_first, e_last} do
    if {y, z} ∈ N_{i-1}^z then
        Add {y, z} to E';
    end
    if {y, z} ∈ N_{i-2}^z then
        if N_{i-2}^z ∩ E_A does not have an edge incident to z then
            Add {y, z} to E';
        end
        if N_{i-2}^z ∩ E_A \ {y, z} has an edge incident to z then
            Let {w, y} be the canonical edge of z incident to y;
            Add {w, y} to E';
        end
    end
end
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 EstimateStretchFactor, 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: EstimateStretchFactor(P, G)

```
1  DT ← L2-DelaunayTriangulation(P);
2  t ← 1;
3  foreach p ∈ P do
4      h ← 1, t_p ← 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 p;
7          t' ← 1;
8          foreach q ∈ X do
9              t' ← max( |π_G(p,q)| / |pq| , t' );
10         end
11         if t' > t_p then
12             h ← h + 1; t_p ← t';
13         else
14             break;
15     end
16     t ← 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 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_∞. For constructing L_2 and L_∞-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 $\frac{1}{2}$-Θ 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 $\lceil 0.7n\rceil \times \lceil 0.7n\rceil$ 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`.

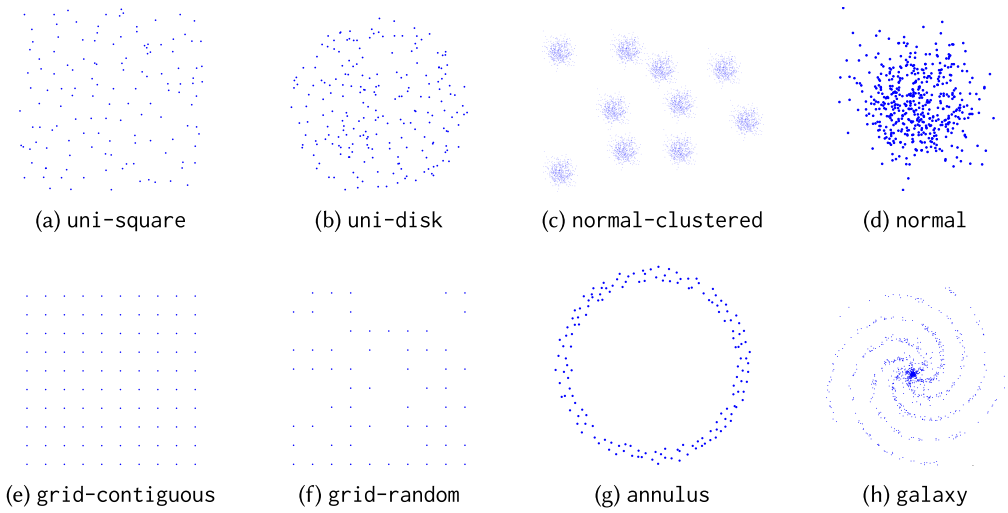

Fig. 2. The eight distributions used to generate synthetic pointsets for our experiments.

(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].

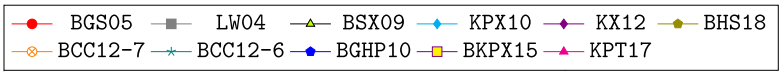

Fig. 3. The plot legends.

- `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 ≈ 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

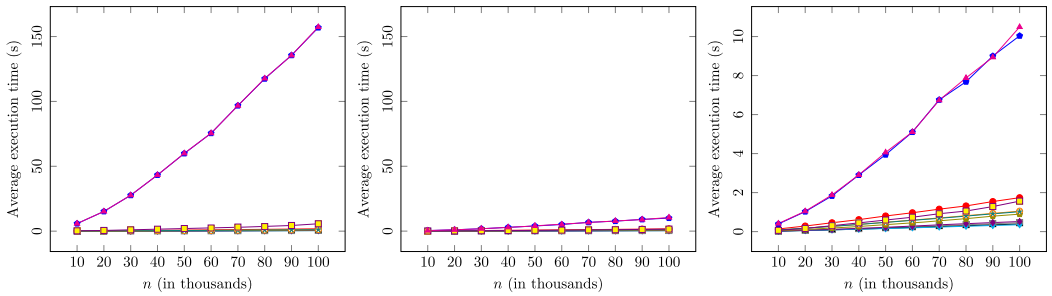

Fig. 4. Points are generated using the uni-square distribution. Left: The running times for all the algorithms are shown; the plots for BGHP10 and KPT17 have overlapped in this figure, and they are the slowest ones in this case. Middle: Here, we consider the runtimes without the time taken to construct 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 BKPX15 was much slower than the others. This is mainly due to the time taken to construct L_∞-Delaunay triangulation. Among the ones that use L_2-Delaunay triangulations, BGS05 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, Δ denotes the theoretical degree upper bound, as claimed by the authors of these algorithms; max Δ_{observed} denotes the maximum degree observed in our experiments; avg Δ_{observed} denotes the observed average degree; and avg Δ_{vertex} denotes the observed average degree per vertex. In our experiments, we found that spanners generated by BGS05, 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

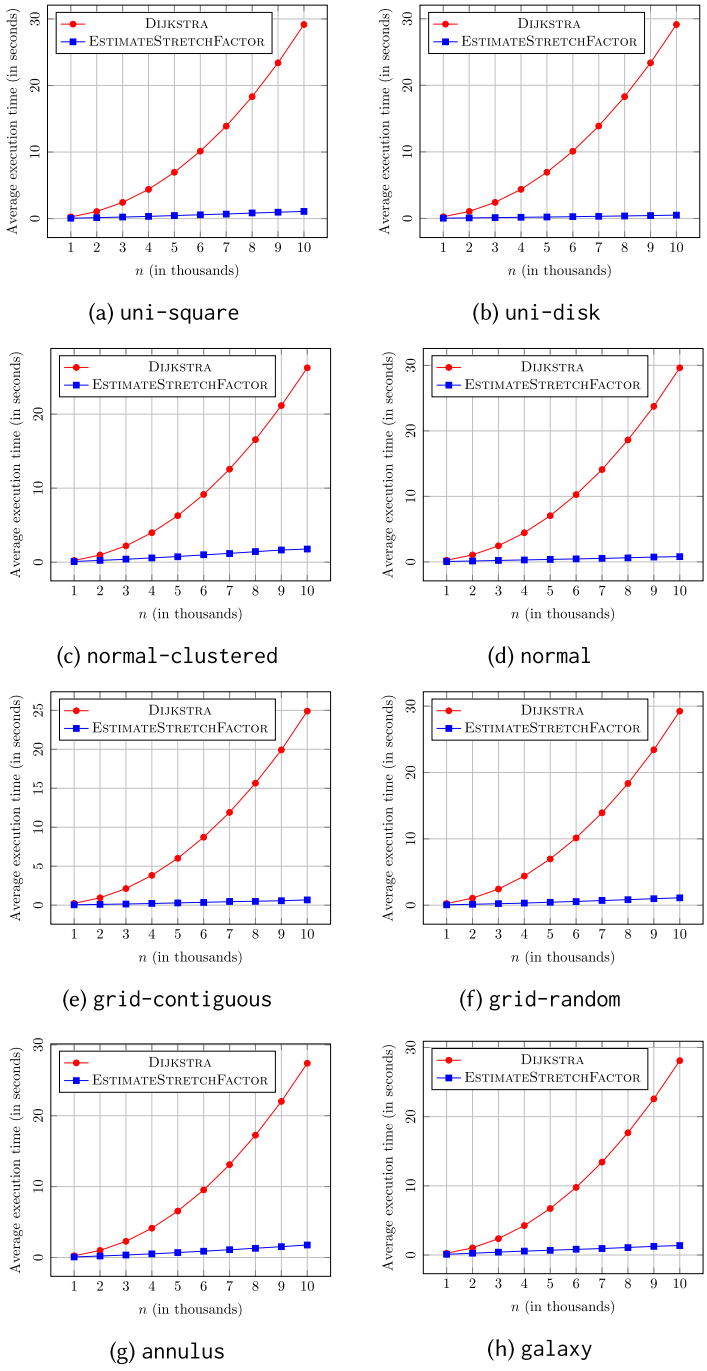

Fig. 5. Runtime comparison: Dijkstra (run from every vertex) vs EstimateStretchFactor. For every value of n, we have used $11 \cdot 5 = 55$ spanner samples since there are 11 algorithms and five pointsets were generated for that value of n using the same distribution.

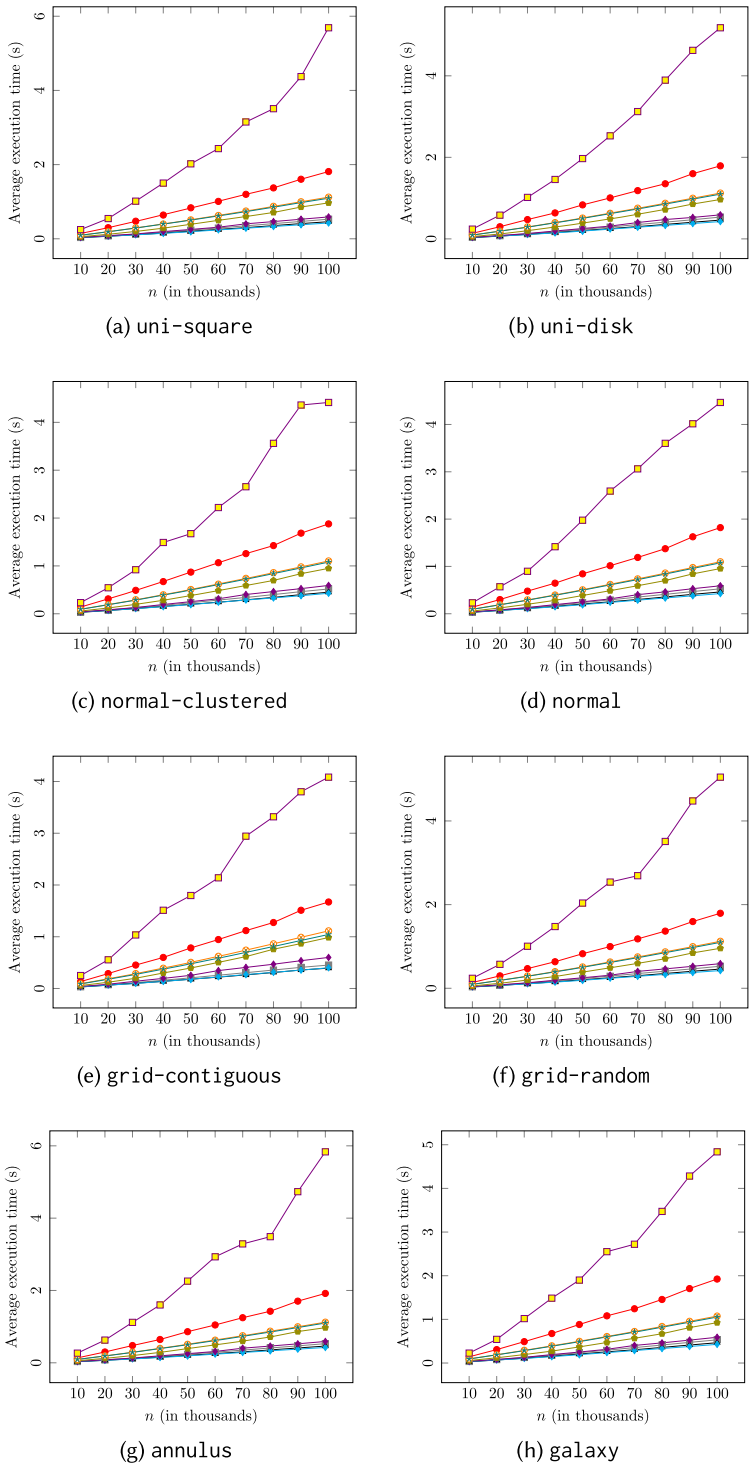

Fig. 6. Runtime comparisons of the nine algorithms (BGHP10 and KPT17 are excluded).

Algorithm	Δ	max Δ_{observed}	avg Δ_{observed}	avg Δ_{vertex}
BGS05	27	13	11.580	5.745
LW04	23	11	9.820	5.148
BSX09	17	9	7.840	4.392
KPX10	14	14	12.980	5.994
KX12	11	10	9.960	5.449
BHS18	8	7	6.980	4.113
BCC12-7	7	7	7.000	4.337
BCC12-6	6	7	6.820	4.016
BGHP10	6	6	6.000	4.240
BKPX15	4	4	4.000	3.328
KPT17	4	4	4.000	3.140

(a) uni-square

Algorithm	Δ	max Δ_{observed}	avg Δ_{observed}	avg Δ_{vertex}
BGS05	27	12	11.540	5.744
LW04	23	11	9.880	5.146
BSX09	17	9	7.760	4.389
KPX10	14	13	12.420	5.993
KX12	11	10	9.980	5.451
BHS18	8	7	6.200	4.109
BCC12-7	7	7	7.000	4.335
BCC12-6	6	7	6.020	4.012
BGHP10	6	6	6.000	4.239
BKPX15	4	4	4.000	3.319
KPT17	4	4	4.000	3.141

(b) uni-disk

Algorithm	Δ	max Δ_{observed}	avg Δ_{observed}	avg Δ_{vertex}
BGS05	27	13	11.600	5.750
LW04	23	11	9.800	5.154
BSX09	17	9	7.860	4.392
KPX10	14	14	12.820	5.996
KX12	11	10	9.980	5.450
BHS18	8	7	6.100	4.110
BCC12-7	7	7	7.000	4.334
BCC12-6	6	6	6.000	4.010
BGHP10	6	6	6.000	4.245
BKPX15	4	4	4.000	3.327
KPT17	4	4	4.000	3.144

(c) normal-clustered

Algorithm	Δ	max Δ_{observed}	avg Δ_{observed}	avg Δ_{vertex}
BGS05	27	13	11.740	5.751
LW04	23	11	9.760	5.158
BSX09	17	9	7.880	4.395
KPX10	14	14	12.320	5.997
KX12	11	10	9.940	5.452
BHS18	8	7	6.060	4.117
BCC12-7	7	7	7.000	4.346
BCC12-6	6	6	6.000	4.023
BGHP10	6	6	6.000	4.254
BKPX15	4	4	4.000	3.327
KPT17	4	4	4.000	3.150

(d) normal

Algorithm	Δ	max Δ_{observed}	avg Δ_{observed}	avg Δ_{vertex}
BGS05	27	11	11.000	5.919
LW04	23	9	7.920	5.099
BSX09	17	9	7.660	4.412
KPX10	14	12	11.880	5.993
KX12	11	11	10.000	5.987
BHS18	8	7	7.000	4.825
BCC12-7	7	7	7.000	5.160
BCC12-6	6	7	6.960	4.361
BGHP10	6	6	6.000	4.979
BKPX15	4	4	4.000	3.350
KPT17	4	4	4.000	3.537

(e) grid-contiguous

Algorithm	Δ	max Δ_{observed}	avg Δ_{observed}	avg Δ_{vertex}
BGS05	27	13	11.540	5.745
LW04	23	11	9.900	5.148
BSX09	17	9	7.980	4.391
KPX10	14	14	12.940	5.995
KX12	11	10	9.980	5.449
BHS18	8	7	6.960	4.114
BCC12-7	7	7	7.000	4.338
BCC12-6	6	7	6.740	4.016
BGHP10	6	6	6.000	4.240
BKPX15	4	4	4.000	3.327
KPT17	4	4	4.000	3.140

(f) grid-random

Algorithm	Δ	max Δ_{observed}	avg Δ_{observed}	avg Δ_{vertex}
BGS05	27	13	11.520	5.730
LW04	23	11	9.740	5.130
BSX09	17	9	8.100	4.382
KPX10	14	14	13.180	5.987
KX12	11	11	10.300	5.448
BHS18	8	7	6.960	4.098
BCC12-7	7	7	7.000	4.313
BCC12-6	6	8	6.940	3.994
BGHP10	6	6	6.000	4.230
BKPX15	4	4	4.000	3.315
KPT17	4	4	4.000	3.130

(g) annulus

Algorithm	Δ	max Δ_{observed}	avg Δ_{observed}	avg Δ_{vertex}
BGS05	27	14	12.320	5.736
LW04	23	11	9.920	5.134
BSX09	17	9	8.180	4.384
KPX10	14	14	13.680	5.993
KX12	11	11	10.000	5.434
BHS18	8	7	6.580	4.090
BCC12-7	7	7	7.000	4.290
BCC12-6	6	7	6.120	3.970
BGHP10	6	6	6.000	4.228
BKPX15	4	4	4.000	3.326
KPT17	4	4	4.000	3.131

(h) galaxy

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)

Algorithm	t	$t_{\max}$	t_{avg}
BGS05	8.3	2.687	2.215
LW04	6.4	2.687	2.349
BSX09	23.6	4.284	3.666
KPX10	2.9	1.519	1.453
KX12	5.7	2.021	1.848
BHS18	4.4	2.812	2.548
BCC12-7	11.6	2.433	2.152
BCC12-6	81.7	2.894	2.494
BGHP10	6	3.204	2.755
BKPX15	156.8	4.692	3.505
KPT17	20	4.895	4.067

Algorithm	ℓ
BGS05	10.209
LW04	4.366
BSX09	3.585
KPX10	5.378
KX12	4.728
BHS18	3.315
BCC12-7	3.636
BCC12-6	3.298
BGHP10	3.429
BKPX15	4.824
KPT17	2.261

(a) uni-square

Algorithm	t	$t_{\max}$	t_{avg}
BGS05	8.3	2.409	2.202
LW04	6.4	2.525	2.325
BSX09	23.6	4.267	3.656
KPX10	2.9	1.497	1.450
KX12	5.7	2.034	1.857
BHS18	4.4	3.152	2.587
BCC12-7	11.6	2.460	2.167
BCC12-6	81.7	2.998	2.460
BGHP10	6	3.034	2.782
BKPX15	156.8	4.255	3.501
KPT17	20	5.050	4.052

Algorithm	ℓ
BGS05	10.014
LW04	4.310
BSX09	3.528
KPX10	5.279
KX12	4.661
BHS18	3.275
BCC12-7	3.597
BCC12-6	3.261
BGHP10	3.415
BKPX15	4.702
KPT17	2.259

(b) uni-disk

Algorithm	t	$t_{\max}$	t_{avg}
BGS05	8.3	2.555	2.214
LW04	6.4	2.522	2.327
BSX09	23.6	4.226	3.664
KPX10	2.9	1.489	1.448
KX12	5.7	2.126	1.856
BHS18	4.4	3.098	2.571
BCC12-7	11.6	2.457	2.161
BCC12-6	81.7	2.814	2.476
BGHP10	6	3.400	2.804
BKPX15	156.8	3.967	3.500
KPT17	20	4.852	4.032

Algorithm	ℓ
BGS05	12.802
LW04	5.220
BSX09	4.345
KPX10	6.752
KX12	5.611
BHS18	3.726
BCC12-7	4.031
BCC12-6	3.680
BGHP10	3.870
BKPX15	5.328
KPT17	2.545

(c) normal-clustered

Algorithm	t	$t_{\max}$	t_{avg}
BGS05	8.3	2.504	2.198
LW04	6.4	2.575	2.329
BSX09	23.6	4.172	3.701
KPX10	2.9	1.500	1.450
KX12	5.7	2.493	1.872
BHS18	4.4	3.090	2.538
BCC12-7	11.6	2.438	2.173
BCC12-6	81.7	2.737	2.447
BGHP10	6	3.178	2.731
BKPX15	156.8	4.344	3.507
KPT17	20	5.236	3.990

Algorithm	ℓ
BGS05	10.111
LW04	4.345
BSX09	3.553
KPX10	5.319
KX12	4.702
BHS18	3.288
BCC12-7	3.603
BCC12-6	3.264
BGHP10	3.431
BKPX15	4.748
KPT17	2.267

(d) normal

Algorithm	t	$t_{\max}$	t_{avg}
BGS05	8.3	3.000	2.812
LW04	6.4	3.000	2.965
BSX09	23.6	3.000	3.000
KPX10	2.9	1.414	1.414
KX12	5.7	1.414	1.414
BHS18	4.4	1.414	1.414
BCC12-7	11.6	1.414	1.414
BCC12-6	81.7	1.414	1.414
BGHP10	6	1.414	1.414
BKPX15	156.8	7.242	6.337
KPT17	20	3.000	3.000

Algorithm	ℓ
BGS05	6.911
LW04	2.820
BSX09	2.455
KPX10	3.506
KX12	3.480
BHS18	2.613
BCC12-7	2.850
BCC12-6	2.286
BGHP10	2.705
BKPX15	3.852
KPT17	1.915

(e) grid-contiguous

Algorithm	t	$t_{\max}$	t_{avg}
BGS05	8.3	2.413	2.199
LW04	6.4	2.707	2.364
BSX09	23.6	4.080	3.686
KPX10	2.9	1.508	1.452
KX12	5.7	2.153	1.838
BHS18	4.4	2.905	2.555
BCC12-7	11.6	2.349	2.126
BCC12-6	81.7	2.698	2.441
BGHP10	6	3.171	2.807
BKPX15	156.8	4.056	3.496
KPT17	20	4.777	4.036

Algorithm	ℓ
BGS05	10.213
LW04	4.367
BSX09	3.585
KPX10	5.382
KX12	4.730
BHS18	3.318
BCC12-7	3.637
BCC12-6	3.300
BGHP10	3.431
BKPX15	4.825
KPT17	2.261

(f) grid-random

Algorithm	t	$t_{\max}$	t_{avg}
BGS05	8.3	2.490	2.185
LW04	6.4	2.735	2.344
BSX09	23.6	4.294	3.668
KPX10	2.9	1.557	1.522
KX12	5.7	2.078	1.862
BHS18	4.4	3.286	2.559
BCC12-7	11.6	2.574	2.162
BCC12-6	81.7	2.922	2.466
BGHP10	6	3.077	2.756
BKPX15	156.8	4.226	3.527
KPT17	20	4.637	4.002

Algorithm	ℓ
BGS05	10.375
LW04	4.445
BSX09	3.675
KPX10	5.524
KX12	4.833
BHS18	3.351
BCC12-7	3.661
BCC12-6	3.321
BGHP10	6.154
BKPX15	7.273
KPT17	3.544

(g) annulus

Algorithm	t	$t_{\max}$	t_{avg}
BGS05	8.3	2.452	2.209
LW04	6.4	2.673	2.323
BSX09	23.6	4.077	3.716
KPX10	2.9	1.512	1.451
KX12	5.7	2.552	1.878
BHS18	4.4	3.147	2.587
BCC12-7	11.6	2.352	2.149
BCC12-6	81.7	2.803	2.483
BGHP10	6	3.309	2.773
BKPX15	156.8	6.833	3.615
KPT17	20	4.867	4.049

Algorithm	ℓ
BGS05	12.119
LW04	5.114
BSX09	4.250
KPX10	6.521
KX12	5.326
BHS18	3.686
BCC12-7	3.949
BCC12-6	3.608
BGHP10	4.398
BKPX15	5.806
KPT17	2.871

(h) galaxy

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

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

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	99999	9.507	2.362	1.970	1.981	2.754	4.340	4.725	4.675	611.018	21.689	613.265
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	168435	15.015	4.310	3.642	3.643	5.142	8.176	8.617	8.507	1494.205	38.482	1498.644
wiki	406648	47.605	11.201	9.479	9.529	13.368	23.072	22.300	22.054	6031.610	101.816	6042.090
vlsi	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	88.983	23.142	19.665	20.071	28.293	51.278	50.648	49.943	16886.620	470.077	16913.390
world	1904711	259.147	57.657	52.078	52.337	74.570	125.078	128.316	125.787	58707.7	1358.70	58780.3
nyctaxi	2728717	376.925	82.787	70.967	72.258	103.779	180.540	180.692	177.558	78800.200	2333.237	78913.100

Fig. 9. 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	99999	12	10	8	13	10	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	168435	12	10	8	13	10	6	7	6	6	4	4
wiki	406648	13	11	9	14	11	7	7	7	6	4	4
vlsi	744710	15	11	9	14	10	7	7	7	6	4	4
china	808693	13	11	9	14	10	6	7	7	6	4	4
world	1904711	14	11	9	14	11	7	7	7	6	4	4
nyctaxi	2728717	15	11	9	14	10	7	7	6	6	4	4

Fig. 10. 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	99999	5.752	5.171	4.428	5.997	5.450	4.111	4.324	4.004	4.246	3.328	3.146
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	168435	5.743	5.161	4.427	5.997	5.436	4.087	4.308	3.989	4.233	3.325	3.136
wiki	406648	5.692	5.088	4.391	5.988	5.361	3.971	4.079	3.787	4.106	3.313	3.043
vlsi	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	5.769	5.213	4.463	5.996	5.520	4.242	4.507	4.154	4.367	3.344	3.211
world	1904711	5.748	5.171	4.438	5.991	5.489	4.151	4.371	4.020	4.318	3.344	3.168
nyctaxi	2728717	5.743	5.148	4.395	5.996	5.436	4.088	4.295	3.975	4.234	3.324	3.133

Fig. 11. Average degree per vertex.

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; $t_{\max}$ denotes the maximum stretch-factor observed in our experiments; and t_{avg} denotes the average observed stretch-factor. Among the 11 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 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 KPX17 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

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	99999	2.552	2.283	3.520	1.438	1.969	2.475	2.253	2.557	2.933	3.519	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	168435	2.245	2.294	4.186	1.479	1.996	2.745	2.185	2.659	3.103	5.192	4.642
wiki	406648	2.400	2.492	3.890	1.495	2.085	2.859	2.300	2.542	2.999	4.392	4.556
vlsi	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	2.356	2.550	3.848	1.482	2.021	2.990	2.277	2.644	3.042	4.136	4.661
world	1904711	2.989	2.961	4.228	1.522	1.997	3.056	2.357	2.657	3.545	6.140	5.422
nyctaxi	2728717	2.462	2.602	4.125	1.468	2.254	3.148	2.344	2.843	3.268	20.009	4.898

Fig. 12. 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	99999	10.662	4.541	3.740	5.601	4.896	3.389	3.703	3.361	3.567	4.986	2.347
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	168435	10.376	4.391	3.690	5.458	4.835	3.228	3.540	3.199	3.389	4.745	2.218
wiki	406648	11.945	5.037	4.156	6.448	5.321	3.585	3.846	3.505	3.827	5.417	2.463
vlsi	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	9.883	4.314	3.564	5.233	4.603	3.287	3.595	3.248	3.481	4.719	2.279
world	1904711	11.145	4.744	3.923	5.917	5.003	3.476	3.777	3.432	3.967	5.272	2.541
nyctaxi	2728717	11.523	5.047	4.145	6.197	5.119	3.144	3.347	2.827	3.208	5.136	2.019

Fig. 13. Lightness of the spanners.

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 ℓ. 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 BGXS05 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 $3n$ 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 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

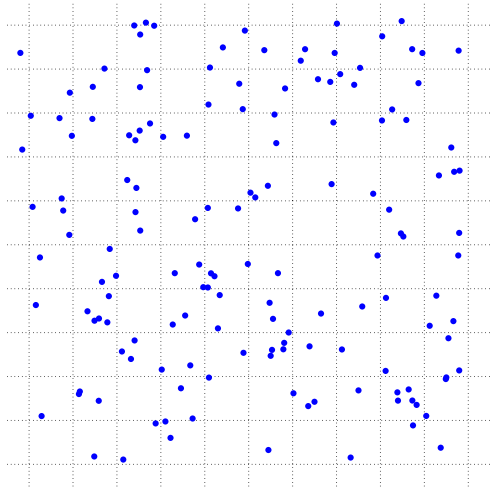

Fig. A.1. A 150-element pointset, drawn randomly from a square.

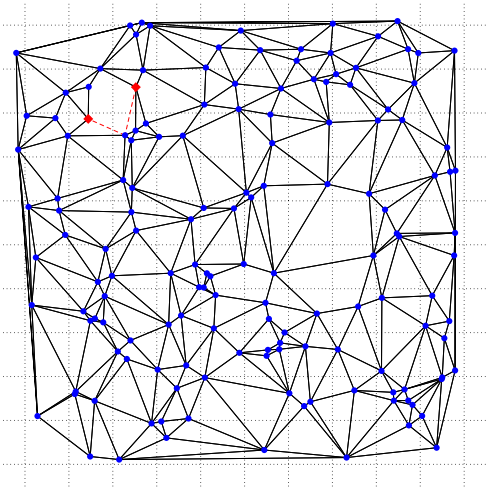

Fig. A.2. The spanner generated by BGS05 on the pointset shown in Figure A.1; degree: 8, stretch-factor: 1.565763.

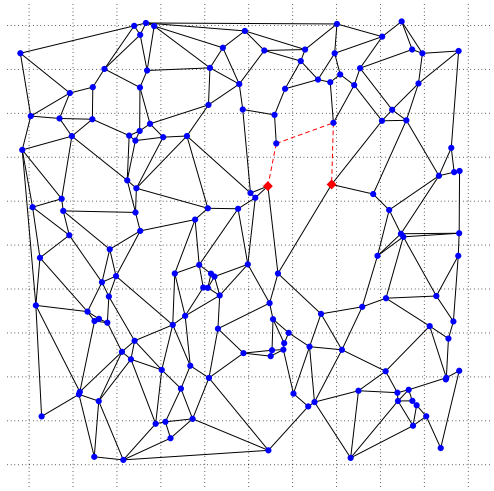

Fig. A.3. The spanner generated by LW04 on the pointset shown in Figure A.1; degree: 6, stretch-factor: 2.602559.

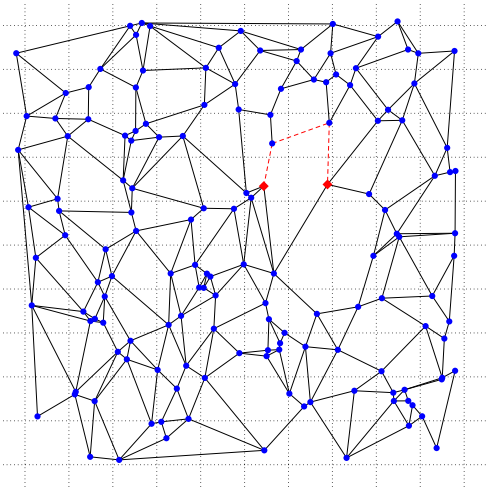

Fig. A.4. The spanner generated by BSX09 on the pointset shown in Figure A.1; degree: 6, stretch-factor: 2.602559.

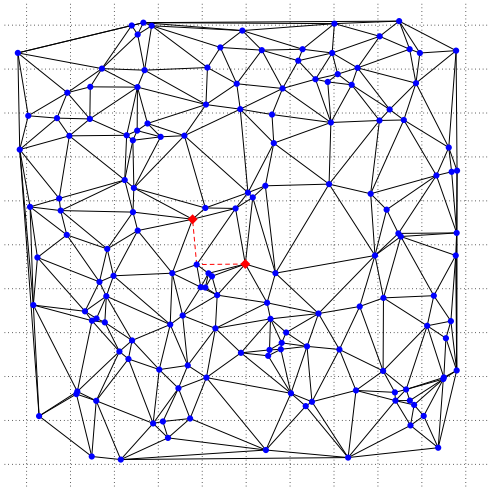

Fig. A.5. The spanner generated by KPX10 on the pointset shown in Figure A.1; degree: 9, stretch-factor: 1.360771.

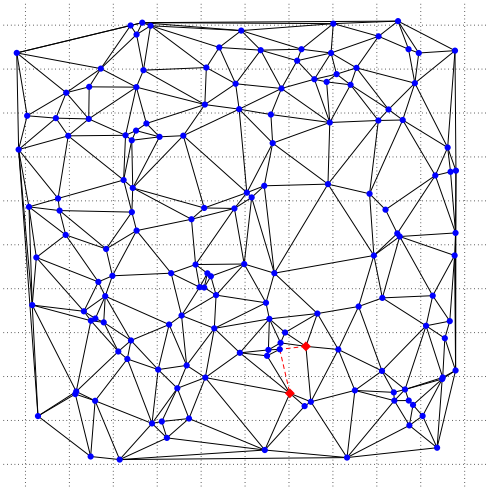

Fig. A.6. The spanner generated by KX12 on the pointset shown in Figure A.1; degree: 8, stretch-factor: 1.440861.

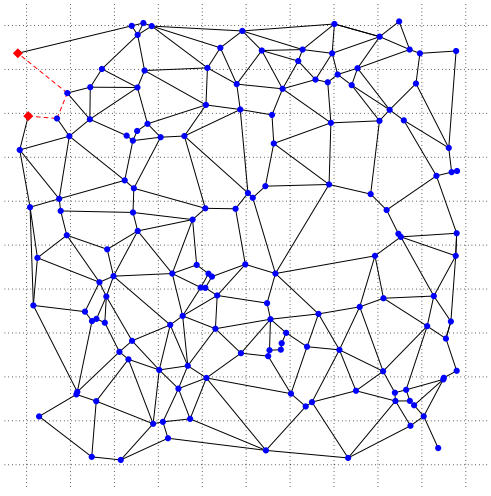

Fig. A.7. The spanner generated by BHS18 on the pointset shown in Figure A.1; degree: 6, stretch-factor: 1.879749.

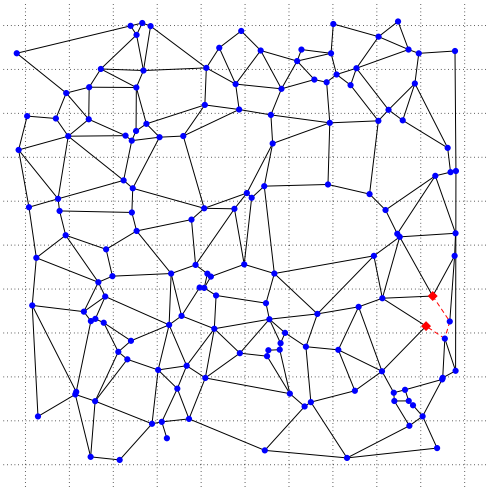

Fig. A.8. The spanner generated by BCC12-7 on the pointset shown in Figure A.1; degree: 6, stretch-factor: 2.302473.

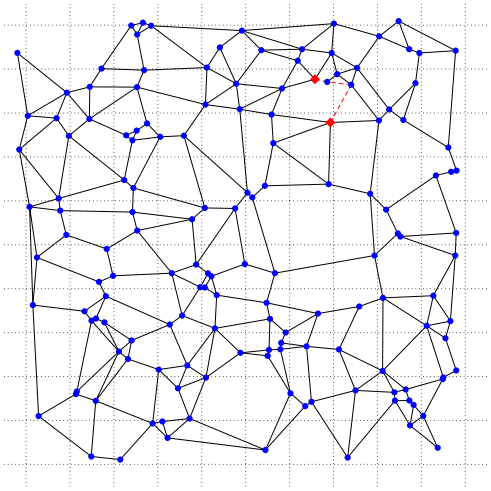

Fig. A.9. The spanner generated by BCC12-6 on the pointset shown in Figure A.1; degree: 6, stretch-factor: 1.735716.

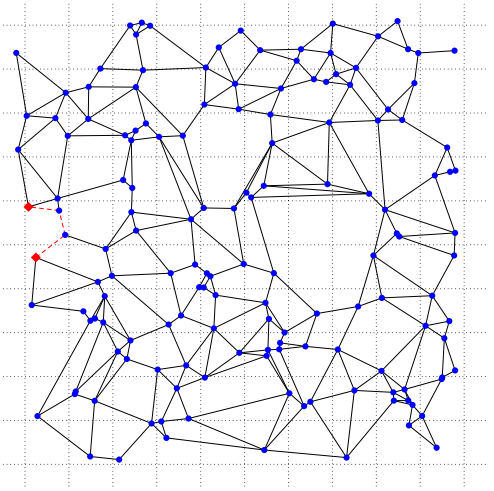

Fig. A.10. The spanner generated by BGHP10 on the pointset shown in Figure A.1; degree: 6, stretch-factor: 1.817045.

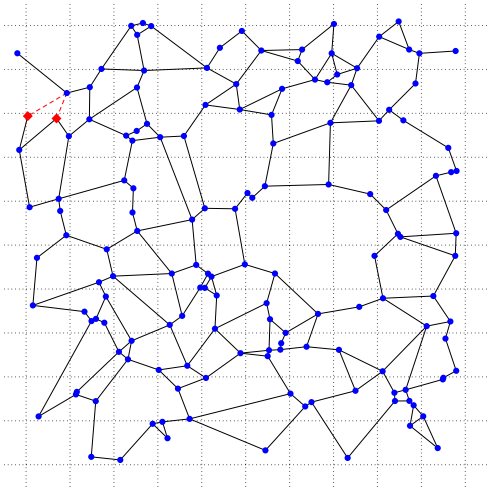

Fig. A.11. The spanner generated by BKPX15 on the pointset shown in Figure A.1; degree: 4, stretch-factor: 2.525204.

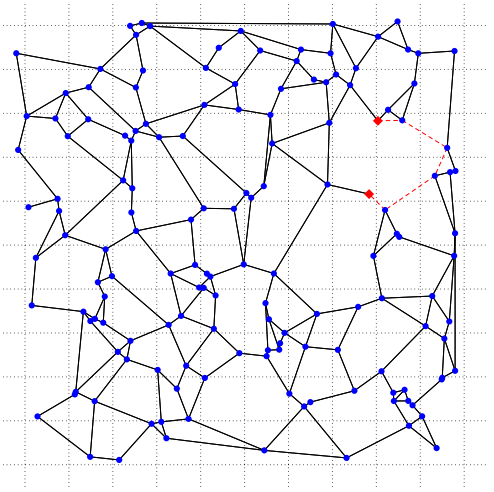

Fig. A.12. The spanner generated by KPT17 on the pointset shown in Figure A.1; degree: 4, stretch-factor: 2.582846.

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.

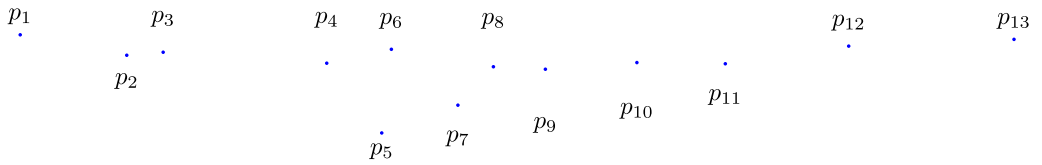

Fig. A.13. A set P of 13 points $p_1, \ldots, p_{13}$. p_1: $(-4.98845, 0.22414)$, p_2: $(-4.23759, 0.08)$, p_3: $(-3.98106, 0.10125)$, p_4: $(-2.82831, 0.02396)$, p_5: $(-2.44066, -0.46761)$, p_6: $(-2.37275, 0.12191)$, p_7: $(-1.90395, -0.27187)$, p_8: $(-1.65373, -0.00109)$, p_9: $(-1.28739, -0.01854)$, 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 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.

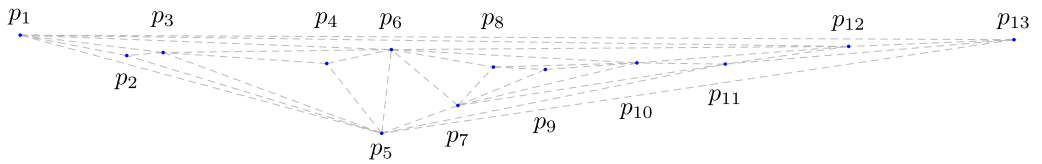

Fig. A.14. The L_2-Delaunay triangulation of P.

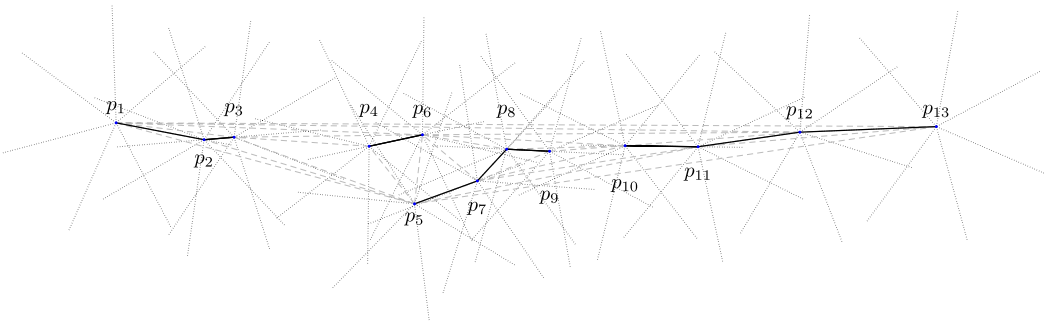

Fig. A.15. The cones (dotted) of each point in P with $\alpha = 2\pi/7$, oriented by the shortest edge incident on that point (bold).

Next, in Figure A.16, we show the edges added by the main portion of the algorithm (excluding the edges added by $\texttt{Wedge}_6$ calls). Only $\texttt{Wedge}_6(p_1, p_2)$ and $\texttt{Wedge}_6(p_{12}, p_{11})$ 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.

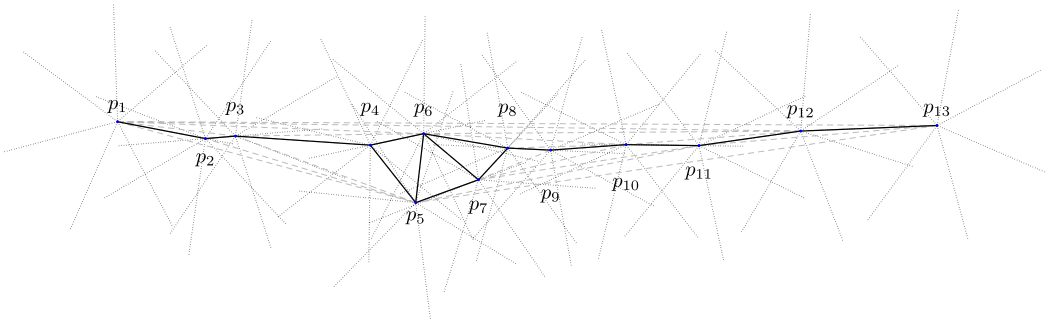

Fig. A.16. Edges added by the main portion of BCC12 (excluding calls to subroutine $\texttt{Wedge}_6$).

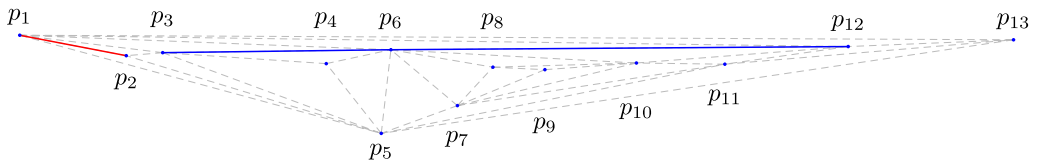

Fig. A.17. The edge p_1p_2 (shown in red) is added during the main portion of the algorithm, and the call to $\texttt{Wedge}_6(p_1, p_2)$ adds the two blue edges p_3p_6 and p_6p_{12}.

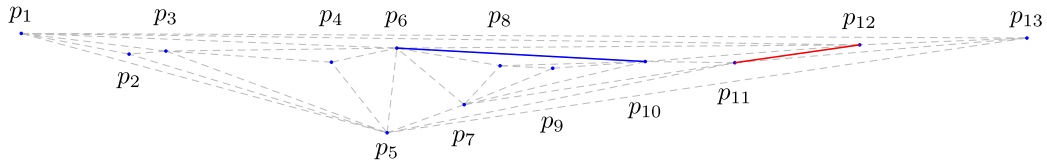

Fig. A.18. The edge $p_{12}p_{11}$ (shown in red) is added during the main portion of the algorithm, and the call to $\texttt{Wedge}_6(p_{12}, p_{11})$ adds the blue edge p_6p_{10}.

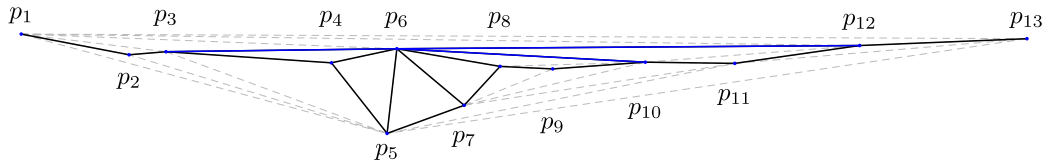

Fig. A.19. The resulting graph on P is a degree-7 plane spanner due to p_6 whose degree is exactly 7. Note that this graph contains the edges shown in Figure A.16 along with the blue edges shown in Figures A.17 and A.18.

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 Jouis 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 3. *Journal of Computational Geometry* 8, 1 (2017), 11–31.

[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_∞-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*.

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

[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 Algorithmics* 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 pointsets. *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.
[32] Iyad Kanj, Ljubomir Perkovic, and Duru Türkoğlu. 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.
[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/.
[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