Dilation-Optimal Edge Deletion in Polygonal Cycles*

Hee-Kap Ahn† Mohammad Farshi‡¶ Christian Knauer§ Michiel Smid¶
Yajun Wang‖

Abstract

Consider a geometric network G in the plane. The dilation between any two vertices x and y in G is the ratio of the shortest path distance between x and y in G to the Euclidean distance between them. The maximum dilation over all pairs of vertices in G is called the dilation of G. In this paper, given a polygonal cycle C on n vertices in the plane, a randomized algorithm is presented which computes in $O(n \log^3 n)$ expected time, the edge of C whose removal results in a polygonal path of smallest possible dilation. It is also shown that the edge whose removal gives a polygonal path of largest possible dilation can be computed in $O(n \log n)$ time. If C is a convex polygon, the running time for the latter problem becomes $O(n)$. Finally, it is shown that for each edge e of C, a $(1-\epsilon)$-approximation to the dilation of the path $C \setminus \{e\}$ can be computed in $O(n \log n)$ total time.

1 Introduction

A *geometric network* on a set V of points in d-dimensional space is a weighted undirected graph $G(V, E)$ with vertex set V whose edges are straight-line segments connecting pairs of points in V and the weight of each edge is the Euclidean distance between its endpoints. Geometric networks naturally model many real-life networks, such as road networks, telecommunication networks, and so on.

Given a (geometric) network, a natural question to ask is what happens to the quality of the network when some connections are removed. In case some links in a traffic network have to be shut down (e.g., due to budget considerations), we may want to know which edges of the network should be removed so as to not decrease the quality of the new network too much. Alternatively, we may want to know the most critical edge in the network, i.e., the edge whose removal causes the largest possible decrease in the quality of the new network.

*Part of this work was done during the Korean Workshop on Computational Geometry at Schloß Dagstuhl in 2006. Work by Ahn was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD, Basic Research Promotion Fund) (KRF-2007-331-D00372). Work by Farshi was supported by Ministry of Science, Research and Technology of I. R. Iran. Work by Smid was supported by NSERC.

†Department of Computer Science and Engineering, POSTECH, Pohang, Korea. heekap@postech.ac.kr

‡Department of Mathematics and Computing Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands. mfarshi@cg.scs.carleton.ca

§Institut für Informatik, Freie Universität Berlin, Takustraße 9, D–14195 Berlin, Germany. christian.knauer@inf.fu-berlin.de

¶School of Computer Science, Carleton University, Ottawa, Ontario, K1S 5B6, Canada. michiel@scs.carleton.ca

‖Department of Computer Science and Engineering, HKUST, Hong Kong S.A.R, China. yalding@cse.ust.hk

We measure the quality of a network G by its *dilation* (or stretch factor) δ_G. The dilation between two distinct *vertices* x and y of the graph G is defined as

$$\delta_G(x, y) := \frac{d_G(x, y)}{|xy|},$$

where $d_G(x, y)$ denotes the distance between x and y in the graph G, that is, the length of the (weighted) shortest path between them, and $|xy|$ denotes the Euclidean distance between x and y. For convenience we define $\delta_G(x, x) := 1$. The dilation between two *sets* X and Y of *vertices* of G is defined as

$$\delta_G(X, Y) := \max\{\delta_G(x, y) \mid x \text{ is a vertex of } X,\ y \text{ is a vertex of } Y\},$$

the dilation of a *set* X of *vertices* of G is defined as

$$\delta_G(X) := \delta_G(X, X),$$

and the *dilation* of the network $G(V, E)$ is defined as

$$\delta_G := \delta_G(V) = \max\{\delta_G(x, y) \mid x \text{ and } y \text{ are vertices of } G\}.$$

For an overview of work on geometric networks, we refer to the recent book by Narasimhan and Smid [17].

Recently the problem of constructing minimum-dilation networks on a given point set has attracted a lot of attention. Klein and Kutz [13] showed that constructing a geometric network on a point set in the plane using a given number of edges such that the network has minimum dilation is NP-hard. The problem is NP-hard even for more specific cases like minimum dilation spanning tree or minimum dilation path, see [4] and [11]. Minimum dilation stars is the case which we can construct in polynomial time, see [7].

All the above problems want to construct a network from scratch but in many applications we already have a network and the problem at hand is to extend/prune the network such that the dilation of the resulting network is minimized. In the case of adding an edge, the optimal edge, i.e. the new edge which minimize the dilation of the network, can be computed in $O(n^4)$ time, where n is the number of nodes in the network. There are also several approximation algorithms which run faster, see [8]. However, the problem of computing the edge in the network whose removal minimizes/maximizes the dilation of the resulting network was not studied before, to the best of our knowledge. We consider a simple variant of this problem: The initial network is a polygonal cycle C in the plane, and we have to remove one single edge from C.

The problem we consider is the following: We are given a polygonal cycle $C = (p_0, \ldots, p_{n-1}, p_0)$ whose n vertices $p_0, \ldots, p_{n-1}$ are points in the plane. We want to determine the edge e of C for which the dilation of the polygonal path $C \setminus \{e\}$ is minimized or maximized. In other words, if we denote by P_i (for $0 \leq i < n$) the polygonal path obtained by removing the edge (p_i, p_{i+1}) from C (where indices are to be read modulo n), then our goal is to compute

$$\delta_C^{\min} := \min_{0 \leq i < n} \delta_{P_i}$$

and

$$\delta_C^{\max} := \max_{0 \leq i < n} \delta_{P_i}.$$

It is known that computing the dilation of a path, or even approximating it, need $\Omega(n \log n)$ time, see [17, Theorem 13.1.3]. Therefore using a naïve algorithm which check all the edges of the cycle takes $O(n^2 \log n)$ time to compute the optimal edge in the cycle.

A summary of our results and the organization of the rest of this paper are in the following. In Section 2, we consider the dilation-minimal edge deletion problem and present a randomized algorithm for the problem. We start in Section 2.1 by describing an approach of [1] to estimate the dilation of a polygonal path. These ideas will play a central role in the algorithm we give in Section 2.2 for solving a decision problem associated with the problem of computing $\delta_C^{\min}$; the algorithm solving this decision problem runs in $O(n \log^2 n)$ expected time. In Section 2.3, we give a simple randomized approach which reduces the problem of computing $\delta_C^{\min}$ to an expected number of $O(\log n)$ decision problems of Section 2.2. Thus, this reduction incurs a logarithmic slowdown of the decision procedure.

In Section 3, we consider the dilation-maximal edge deletion problem and present a $O(n \log n)$ time algorithm for the problem. We first show that for two fixed vertices x and y of C, it is easy to determine the largest possible dilation between them if one edge is removed from C. We then show that, in order to compute $\delta_C^{\max}$, it suffices to consider pairs (x, y) of vertices whose distance is at most twice the closest-pair distance in the vertex set of C. Since there are only $O(n)$ such pairs (x, y), this leads to an efficient algorithm for computing $\delta_C^{\max}$.

Finally, we present an approximation algorithm that computes in $O(n \log n)$ total time the dilation of each path P_i, as well as an approximation of $\delta_C^{\min}$ in Section 4. The algorithm uses the well-separated pair decomposition of [3] and a result of [16], which states that this decomposition can be used to reduce the problem of approximating the dilation of a Euclidean graph to the problem of computing the shortest-path distances between $O(n)$ pairs of vertices. This result, together with the observation that for any two vertices x and y of C, the sequence $\delta_{P_0}(x, y), \ldots, \delta_{P_{n-1}}(x, y)$ contains only two distinct values, leads to an $O(n \log n)$–time algorithm that approximates the dilation of each path P_i as well as the minimum dilation $\delta_C^{\min}$.

2 Dilation-Minimal Edge Deletion in a Polygonal Cycle

In this section, we give an algorithm which given a polygonal cycle C, computes the edge whose removal generates a path which has minimum dilation among all the paths generated by removing an edge from C. The algorithm starts with removing a random edge from C and computes the dilation κ of the generated path. Then it uses a decision algorithm which for each edge e in the cycle C, decides whether the dilation of the path $C \setminus \{e\}$ is less than κ. Then it picks one of the edges whose removal generates a path with dilation less than κ and assigns the dilation of the path to κ. It repeats the procedure until no edge in the cycle generates a path with dilation less than κ.

2.1 Estimating The Dilation of a Polygonal Path

Our algorithm for computing the edge of a polygonal cycle whose removal minimizes the dilation of the resulting path uses as a subroutine parts of the algorithm of [1] that decides if the dilation of a polygonal path is less than some given threshold $\kappa > 1$. We describe those parts of this algorithm which are relevant for us.

Let $P = (p_0, \ldots, p_{n-1})$ be a polygonal path whose n vertices are points in the plane and let $\kappa \geq 1$ be a real number. Without loss of generality we assume p_0 is the origin. The idea is to use a

lifting transformation that rephrases the decision problem, i.e., the problem of deciding if $\delta_P < \kappa$, into a point-cone incidence-problem in $\mathbb{R}^3$.

We denote the first and last vertices of a polygonal path P by $f(P)$ and $l(P)$, respectively. Thus, $f(P) = p_0$. For each vertex p of P, we define the *weight* of p to be

$$\omega_P(p) := d_P(p, f(P))/\kappa.$$

We map each vertex $p = (x_p, y_p)$ of P to the point

$$h_P(p) := (x_p, y_p, \omega_P(p)) \in \mathbb{R}^3.$$

Let $\mathcal{C}$ denote the three-dimensional cone

$$\mathcal{C} := \{(x, y, z) \in \mathbb{R}^3 \mid z = \sqrt{x^2 + y^2}\}.$$

We map each vertex p of P to the cone

$$\mathcal{C}_P(p) := \mathcal{C} \oplus h_P(p) = \{c + h_P(p) \mid c \in \mathcal{C}\}.$$

Note that we can reformulate the definition of $\mathcal{C}_P(p)$ as

$$\mathcal{C}_P(p) := \left\{(x, y, z) \mid z - \omega_P(p) = \sqrt{(x - x_p)^2 + (y - y_p)^2}\right\},$$

and therefore $\mathcal{C}_P(p)$ is the graph of the bivariate function $f_p(q) = |pq| + \omega_P(p)$ for $q \in \mathbb{R}^2$. If p and q are vertices of P, then we say that p is *before* q on P, if $d_P(p, f(P)) < d_P(q, f(P))$; this will be denoted as $p \prec_P q$. We then get the following lemma. (see Figure 1 for an illustration).

Lemma 1 *For any two vertices p and q of P with $p \prec_P q$, we have*

$$\delta_P(p, q) < \kappa \text{ if and only if } h_P(q) \text{ lies below } \mathcal{C}_P(p).$$

Proof. By straightforward algebraic manipulation, we have

$$\begin{aligned}
\delta_P(p, q) < \kappa \quad &\iff \quad \frac{d_P(q, p)}{|qp|} < \kappa \\
&\iff \quad \frac{d_P(f(P), q) - d_P(f(P), p)}{|qp|} < \kappa \\
&\iff \quad \frac{d_P(f(P), q)}{\kappa} < |qp| + \frac{d_P(f(P), p)}{\kappa} \\
&\iff \quad \omega_P(q) < |qp| + \omega_P(p) = f_p(q).
\end{aligned}$$

□

If X and Y are subsets of the vertex set of P, then we say that X is *before* Y on P, if $d_P(x, f(P)) < d_P(y, f(P))$ for all $x \in X$ and all $y \in Y$; this will be denoted as $X \prec_P Y$. For any subset X of the vertex set of P, we define

$$\mathcal{C}_P(X) := \{\mathcal{C}_P(p) \mid p \in X\}$$

and

$$h_P(X) := \{h_P(p) \mid p \in X\}.$$

The lower envelope of a set S of bi-variate functions will be denoted as $\mathcal{L}(S)$. Lemma 1 immediately gives the following result.

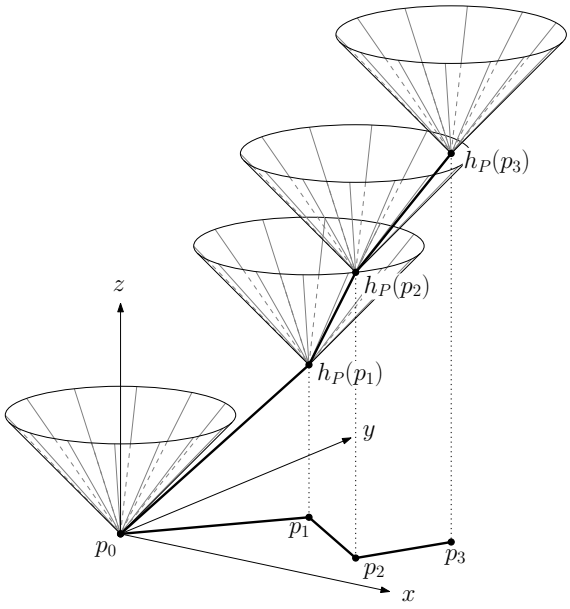

Figure 1: Illustration of the lifting technique.

Lemma 2 *For any two subsets X and Y of the vertex set of P with $X \prec_P Y$, we have*

$$\delta_P(X,Y) < \kappa \text{ if and only if } h_P(Y) \text{ lies below } \mathcal{L}(\mathcal{C}_P(X)).$$

The minimization diagram of $\mathcal{C}_P(X)$, i.e., the projection of the lower envelope $\mathcal{L}(\mathcal{C}_P(X))$ onto the xy-plane, is the additively weighted Voronoi diagram $V_P(X)$ of X with respect to the weight function ω_P. If the point y of Y is located in the Voronoi region of the point x of X, then $h_P(y)$ is below $\mathcal{L}(\mathcal{C}_P(X))$ if and only if $h_P(y)$ is below $\mathcal{C}_P(x)$.

This yields an efficient algorithm to verify if $\delta_P(X,Y) < \kappa$ for two subsets X and Y of the vertex set of P having the property that $X \prec_P Y$: The Voronoi diagram $V_P(X)$ can be computed in $O(|X| \log |X|)$ time, c.f. [9]. Within the same time bound, this diagram can be preprocessed into a linear size data structure that supports $O(\log |X|)$-time point-location queries, c.f. [12]. This structure can now be queried with each point y of Y to determine which point x of X contains y in its Voronoi cell. Once this is known, the check if $h_P(y)$ is below $\mathcal{C}_P(x)$ can be performed in $O(1)$ time. The total running time of this algorithm is $O((|X| + |Y|) \log |X|)$.

2.2 The Decision Problem

Let C be a polygonal cycle on a set of n vertices in the plane and let $\kappa > 1$ be a real number. In this section, we present an algorithm that decides for each edge e of C, whether or not the dilation of the polygonal path $C \setminus \{e\}$ is less than κ. We first describe the overall approach. Then, we give two implementations that yield running times of $O(n \log^3 n)$ and $O(n \log^2 n)$, respectively.

For a cycle $C = (p_0, p_1, \ldots, p_{n-1}, p_0)$, we call p_{i+1} the successor of p_i in C (where indices are to be read modulo n) and denote it by $\mathrm{succ}(p_i)$. If $R = (r_1, \ldots r_m)$ and $Q = (q_1, \ldots, q_n)$ are two polygonal paths having the property that $q_1 = f(Q)$ is the successor of $l(R) = r_m$ in C,

then we denote the concatenation of R and Q by $R \oplus Q$. Thus, $R \oplus Q$ is the polygonal path $(r_1, \ldots, r_m, q_1, \ldots, q_n)$.

In order to facilitate a recursive approach, we will consider the following more general problem: Assume that (the vertex set of) C is partitioned into two polygonal paths T (the *top*) and B (the *bottom*) such that $\delta_T < \kappa$. We want to decide for each edge e of B, i.e., any edge e on the cycle C with both endpoints in B, whether or not the dilation of $C \setminus \{e\}$ is less than κ. If we take $T = \emptyset$ then we obtain the original problem.

Algorithm *DilationDecision*(T, B, κ)
Input: Paths T and B and $\kappa > 1$.
Output: *yes* or *no* for every edge of B.
1. **if** $|B| = 2$
2. **then return** *yes* for the edge in B;
3. $l :=$ the last vertex of B; (∗ `in counterclockwise order along` C ∗)
4. $r :=$ the first vertex of B; (∗ `in counterclockwise order along` C ∗)
5. $m :=$ the middle vertex of B;
6. $B^r :=$ the part of the path B from r to m;
7. $B^l :=$ the part of the path B from succ(m) to l;
8. **if** $\delta_{T \oplus B^r} < \kappa$
9. **then** *DilationDecision*$(T \oplus B^r, B^l, \kappa)$;
10. **else return** *no* for each edge e of B^l;
11. **if** $\delta_{B^l \oplus T} < \kappa$
12. **then** *DilationDecision*$(B^l \oplus T, B^r, \kappa)$;
13. **else return** *no* for each edge e of B^r;

The details of the decision algorithm are presented in Algorithm *DilationDecision*. The correctness of Algorithm *DilationDecision* is obvious. We will show below that after a preprocessing step taking $O(n \log^2 n)$ expected time, we can decide in $O(|B| \log n)$ time if $\delta_{T \oplus B^r} < \kappa$ and $\delta_{B^l \oplus T} < \kappa$, where $|B|$ denotes the number of vertices on B. The expected running time $t(n)$ of the algorithm can therefore be written as $t(n) = O(n \log^2 n) + r(n)$ where the function r satisfies the recurrence

$$r(b) \leq 2 \cdot r(b/2) + O(b \log n).$$

This implies that $t(n) = O(n \log^2 n)$.

Figure 2 illustrates the recursion tree of Algorithm *DilationDecision*. The nodes of the tree are labeled according to a breadth-first search (BFS) numbering where the first (left) child of a node corresponds to the recursive call in Step 12 and the second (right) child corresponds to the recursive call in Step 9. Later, we will refer to a recursive call corresponding to the node with BFS-number i as the i-th step of the recursion. For each node i, the current top and bottom paths are denoted by T_i and B_i, respectively. These paths can be computed as follows. Assume that the polygonal cycle C is given by the array $C[0, \ldots, n]$ and that B consists of b vertices. Then $B_1 = C[0, \ldots, b-1]$ and $T_1 = T$. For $i \geq 1$, if $B_i = C[l, r]$, then $B_{2i} := C[l, l + \lfloor \frac{r-l}{2} \rfloor]$, $B_{2i+1} := C[l + \lfloor \frac{r-l}{2} \rfloor + 1, r]$, $T_{2i} := B_{2i+1} \oplus T_i$, and $T_{2i+1} := T_i \oplus B_{2i}$. Observe that each top path T_j is the concatenation of T_1 and $O(\log n)$ bottom paths.

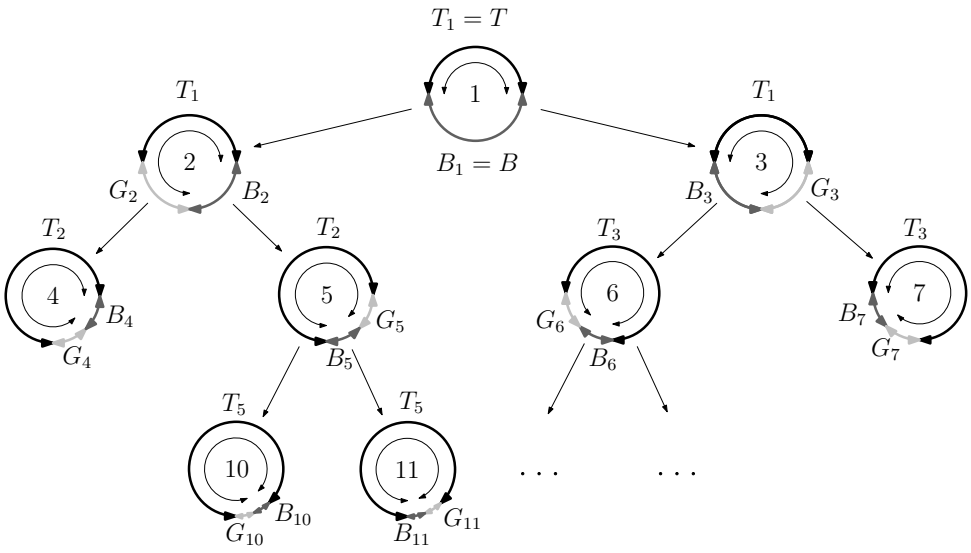

Figure 2: The recursion tree. Note that $G_{2i} := B_{2i+1}$ and $G_{2i+1} := B_{2i}$.

2.2.1 A first implementation.

We will show that after an $O(n \log^2 n)$–time preprocessing, we can decide if (i) $\delta_{T \oplus B^r} < \kappa$ and (ii) $\delta_{B^l \oplus T} < \kappa$ in $O(|B| \log^2 n)$ time. Later, we will give a faster implementation. Since (ii) is symmetric to (i), we only show how to decide whether or not (i) holds.

Suppose we have a polygonal path T' with $\delta_{T'} < \kappa$ that is given as a list of k polygonal paths $(B'_1, \ldots, B'_k)$ such that $f(B'_{i+1})$ is the successor of $l(B'_i)$ in the cycle C, for $1 \leq i < k$. Thus, we have $T' = B'_1 \oplus \ldots \oplus B'_k$. Given a new polygonal chain B' with $f(B') = \text{succ}(l(T'))$ in C, we want to decide if $\delta_{T' \oplus B'} < \kappa$.

Observe that $\delta_{T' \oplus B'} < \kappa$ if and only if (a) $\delta_{T'} < \kappa$, (b) $\delta_{B'} < \kappa$, and (c) $\delta_{T' \oplus B'}(T', B') < \kappa$. We are given that (a) holds. Using the algorithm of [1], we can decide in $O(|B'| \log |B'|)$ expected time whether or not (b) holds. Thus, it remains to show how to verify whether or not (c) holds.

Obviously, $\delta_{T' \oplus B'}(T', B') < \kappa$ if and only if $\delta_{T' \oplus B'}(B'_i, B') < \kappa$ for each i with $1 \leq i \leq k$. Since $B'_i \prec_{T' \oplus B'} B'$, we know from Lemma 2 that $\delta_{T' \oplus B'}(B'_i, B') < \kappa$ if and only if $h_{T' \oplus B'}(B')$ lies below $\mathcal{L}(\mathcal{C}_{T' \oplus B'}(B'_i))$.

Assume that for each path B'_i, we have the total accumulated scaled length

$$\ell_i := \sum_{j=1}^{i} d_{B'_j}(l(B'_j), f(B'_j))/\kappa$$

and the additively weighted Voronoi diagram $V_{B'_i}(B'_i)$ that has been augmented with a data structure to support point-location queries in t_{loc} time per query. Recall that $V_{B'_i}(B'_i)$ is the projection of the lower envelope $\mathcal{L}(\mathcal{C}_{B'_i}(B'_i))$ onto the xy-plane. It is defined with respect to the weights

$$\omega_{B'_i}(p) = d_{B'_i}(p, f(B'_i))/\kappa.$$

Since

$$\omega_{T' \oplus B'}(p) = \omega_{B'_i}(p) + \ell_{i-1}$$

for all $p \in B'_i$, we have

$$\omega_{T' \oplus B'}(p) - \omega_{T' \oplus B'}(q) = \omega_{B'_i}(p) - \omega_{B'_i}(q)$$

for all $p, q \in B'_i$. It follows that the diagram $V_{B'_i}(B'_i)$ is also the projection of the lower envelope $\mathcal{L}(\mathcal{C}_{T' \oplus B'}(B'_i))$.

The associated point-location structure of B'_i can therefore be used to determine for each point b' in B', the point t in B'_i that contains b' in its Voronoi cell in $V_{T' \oplus B'}(B'_i)$. Once this is known for each point b' in B', we can check if $h_{T' \oplus B'}(b')$ is below $\mathcal{C}_{T' \oplus B'}(t)$. To this end, we compute the weights

$$\omega_{T' \oplus B'}(t) = \omega_{B'_i}(t) + \ell_{i-1}$$

and

$$\omega_{T' \oplus B'}(b') = \omega_{B'}(b') + \ell_k.$$

The overall running time of this approach (excluding the preprocessing time) is $O(k|B'|t_{loc})$.

In our application, the relevant paths B'_i are the bottom paths B_i that appear in the recursive calls. As a consequence, $k = O(\log n)$, $|T' \oplus B'| \leq n$, and we can precompute the required information in $O(n \log^2 n)$ time by computing all the diagrams $V_{B_i}(B_i)$ along with the point-location data structures. Since $t_{loc} = O(\log n)$, it follows that the overall running time of this approach (after $O(n \log^2 n)$ preprocessing time) is $O(|B'| \log^2 n)$. With this implementation, Algorithm *Dilation-Decision* runs in $O(n \log^3 n)$ expected time.

2.2.2 A faster implementation.

Consider the i-th node of the recursion tree (according to the BFS-numbering). This node represents a recursive call *DilationDecision* whose second input parameter is the set B_i. In this call, we split B_i (almost evenly) into B_{2i} and B_{2i+1}, and compute the diagrams $V_{B_{2i}}(B_{2i})$ and $V_{B_{2i+1}}(B_{2i+1})$. We then locate each point b of B_{2i} in $V_{B_{2i+1}}(B_{2i+1})$ to determine which point t of B_{2i+1} contains b in its Voronoi cell in $V_{B_{2i+1}}(B_{2i+1})$. We store t in a table $\mathcal{T}_b$ associated with b under the key $2i+1$ that identifies the set B_{2i+1}. In the same way, we locate each point b of B_{2i+1} in $V_{B_{2i}}(B_{2i})$ and store the corresponding point t of B_{2i} in a table $\mathcal{T}_b$ associated with b under the key $2i$.

Since we perform exactly one point-location query for each point b of B_1 on each level of the recursion tree, the table $\mathcal{T}_b$ has $O(\log n)$ entries. We can therefore use the construction of [10] to store $\mathcal{T}_b$ in a perfect-hash table of size $O(\log n)$ that supports $O(1)$ access time. Note that in the complexity model of [10], it is assumed that the entries come from a finite universe and the algorithm is able to randomly access each memory location in constant time. Recall that the construction of [10] is randomized and builds the hash table in $O(\log n)$ expected time.

The total time we spend on each level of the recursion tree is $O(n \log n)$, so the total expected preprocessing time is $O(n \log^2 n)$ and the total time we spend for answering point-location queries is $O(n \log^2 n)$.

In order to determine for a point b' of B', where $B' \subseteq B_1$, which point t of B'_i contains b' in its Voronoi cell, we find the index j for which $B'_i = B_j$. Then we retrieve the entry with the key j from $\mathcal{T}_{b'}$. This is exactly the point t of B_j that contains b in its Voronoi cell in $V_{B_j}(B_j)$.

It follows that $t_{loc} = O(1)$, so that the overall running time of this approach (after $O(n \log^2 n)$ preprocessing time) is $O(|B'| \log n)$. With this implementation, Algorithm *DilationDecision* runs in $O(n \log^2 n)$ expected time.

2.3 The Optimization Algorithm

We now present our algorithm that computes, for a given polygonal cycle C on a set of n points in the plane, the value of $\delta_C^{\min}$ in $O(n \log^3 n)$ expected time. Clarkson and Shor [5] used a similar randomized approach to compute the diameter of a point set.

Step 1: Compute a random permutation of the edges of C. We denote the permutation by $e_1, e_2, \ldots, e_n$.

Step 2: Use the algorithm of [1] to compute the dilation of the path $C \setminus \{e_1\}$ and assign this value to κ.

Step 3: Run Algorithm *DilationDecision* and store with each edge e of C a Boolean flag which is *true* if and only if the dilation of the path $C \setminus \{e\}$ is less than κ.

Step 4: For $i = 2, 3, \ldots, n$, do the following: If the flag stored with e_i is *true*, then perform the following Steps 4.1 and 4.2:

Step 4.1: Use the algorithm of [1] to compute the dilation of the path $C \setminus \{e_i\}$ and assign this value to κ.

Step 4.2: Run Algorithm *DilationDecision* and store with each edge e of C a Boolean flag which is *true* if and only if the dilation of the path $C \setminus \{e\}$ is less than κ.

Step 5: Return the value of κ.

The correctness of the algorithm follows from the fact that, after Step 4,

$$\kappa = \min_{1 \le i \le n} \delta_{P_i} = \delta_C^{\min},$$

where P_i is the polygonal path obtained by removing e_i from C.

Clearly, Step 1 takes $O(n)$ time. The algorithm of [1] and, therefore, Step 2, takes $O(n \log n)$ expected time. Each time we run Algorithm *DilationDecision*, we spend $O(n \log^2 n)$ expected time. Observe that we run this algorithm once in Step 3 and, moreover, in Step 4 each time the dilation of P_i is less than the current value of κ. In the latter case, we also spend $O(n \log n)$ expected time to compute the dilation of P_i. Since the edges of C are in random order, the values $\delta_{P_1}, \ldots, \delta_{P_n}$ are in random order as well. At the start of the i-th iteration of Step 4, the value of κ is equal to $\min_{1 \le j < i} \delta_{P_j}$. Thus, $\delta_{P_i} < \kappa$ if and only if δ_{P_i} is the minimum of the set $\{\delta_{P_j} \mid 1 \le j \le i\}$. It follows that $\delta_{P_i} < \kappa$ with probability $1/i$. Using the linearity of expectation, it follows that the expected number of times that Steps 4.1 and 4.2 are performed is equal to $\sum_{i=2}^{n} 1/i = O(\log n)$. Thus, the overall expected running time of our algorithm is $O(n \log^3 n)$.

Theorem 1 *Given a polygonal cycle C on n vertices in the plane, we can compute $\delta_C^{\min}$ in $O(n \log^3 n)$ expected time.*

3 Dilation-Maximal Edge Deletion in a Polygonal Cycle

Let $C = (p_0, \ldots, p_{n-1}, p_0)$ be a polygonal cycle whose vertices $p_0, \ldots, p_{n-1}$ are points in the plane. Recall from Section 1 that P_i (for $0 \le i < n$) denotes the polygonal path obtained by removing the edge (p_i, p_{i+1}) from C, $d_{P_i}(x, y)$ denotes the length of the subpath of P_i between x and y,

$\delta_{P_i}(x,y) = d_{P_i}(x,y)/|xy|$ denotes the dilation between x and y in P_i, and δ_{P_i} denotes the dilation of P_i. In this section, we give an algorithm that computes

$$\delta_C^{\max} = \max_{0 \le i < n} \delta_{P_i}.$$

Let L be the total length of the edges of C. We define $\Delta(p_0) := 0$ and $\Delta(p_i) := \Delta(p_{i-1}) + |p_{i-1}p_i|$ for $1 \le i < n$. Thus, $\Delta(p_i)$ is the length of the path $(p_0, \ldots, p_i)$ and the shortest-path distance $d_C(p_i, p_j)$ between p_i and p_j in the cycle C is given by

$$d_C(p_i, p_j) = \min(|\Delta(p_i) - \Delta(p_j)|, L - |\Delta(p_i) - \Delta(p_j)|).$$

Consider two distinct vertices x and y of C. We obtain the largest dilation between x and y in any path P_i, by deleting an arbitrary edge on the shorter of the two paths in C between x and y. Thus, the following lemma holds.

Lemma 3 *Let x and y be two distinct vertices of C. Then*

$$\max_{0 \le i < n} \delta_{P_i}(x,y) = \frac{\max(|\Delta(x) - \Delta(y)|, L - |\Delta(x) - \Delta(y)|)}{|xy|} \ge \frac{L}{2|xy|}.$$

The next lemma states that the closest pair in the vertex set of C can be used to obtain a 2-approximation to $\delta_C^{\max}$.

Lemma 4 *Let (p,q) be a closest pair in the vertex set of C. Then*

$$\delta_C^{\max} \le 2 \cdot \max_{0 \le i < n} \delta_{P_i}(p,q).$$

Proof. Let j be an index such that $\delta_C^{\max} = \delta_{P_j}$ and let x and y be two vertices of C such that $\delta_{P_j} = \delta_{P_j}(x,y)$. Then

$$\delta_C^{\max} = \frac{d_{P_j}(x,y)}{|xy|} \le \frac{L}{|pq|}.$$

By Lemma 3, we have

$$\frac{L}{|pq|} \le 2 \cdot \max_{0 \le i < n} \delta_{P_i}(p,q).$$

□

Thus, by computing the closest pair (p,q) in the vertex set of C and then using Lemma 3 to compute $\max_{0 \le i < n} \delta_{P_i}(p,q)$, we obtain a 2-approximation to $\delta_C^{\max}$. We now show that a simple extension leads to an algorithm that computes the exact value of $\delta_C^{\max}$.

Let S be the set of all pairs (x,y) in the vertex set of C for which $x \ne y$ and $|xy| \le 2|pq|$. The following lemma states that it suffices to consider the elements of S to compute $\delta_C^{\max}$.

Lemma 5 *We have*

$$\delta_C^{\max} = \max_{(x,y) \in S} \max_{0 \le i < n} \delta_{P_i}(x,y).$$

Proof. It is clear that

$$\delta_C^{\max} = \max_{x,\,y \text{ vertices of } C} \; \max_{0 \le i < n} \delta_{P_i}(x,y) \ge \max_{(x,y) \in S} \; \max_{0 \le i < n} \delta_{P_i}(x,y).$$

Let j be an index such that $\delta_C^{\max} = \delta_{P_j}$ and let a and b be two vertices of C such that $\delta_{P_j} = \delta_{P_j}(a,b)$. If we can show that $(a,b) \in S$ (i.e., $|ab| \le 2|pq|$), then the proof is complete. By Lemma 3, we have

$$\frac{L}{2|pq|} \le \max_{0 \le i < n} \delta_{P_i}(p,q).$$

It follows that

$$\begin{aligned}
\frac{L}{2|pq|} &\le \max_{0 \le i < n} \delta_{P_i}(p,q) \\
&\le \max_{x,\,y \text{ vertices of } C} \; \max_{0 \le i < n} \delta_{P_i}(x,y) \\
&= \delta_C^{\max} \\
&= \delta_{P_j}(a,b) \\
&= \frac{d_{P_j}(a,b)}{|ab|} \\
&\le \frac{L}{|ab|}.
\end{aligned}$$

This implies that $|ab| \le 2|pq|$. □

The discussion above leads to the following algorithm for computing the value of $\delta_C^{\max}$.

Step 1: Compute the total length L of the cycle C and compute the values $\Delta(p_i)$ $(0 \le i < n)$ as defined above.

Step 2: Compute the closest pair (p,q) in the vertex set of C.

Step 3: Compute the set S of all pairs (x,y) in the vertex set of C for which $x \ne y$ and $|xy| \le 2|pq|$.

Step 4: For each element (x,y) in S, compute

$$\frac{\max(|\Delta(x) - \Delta(y)|,\, L - |\Delta(x) - \Delta(y)|)}{|xy|}.$$

Step 5: Return the largest value computed in Step 4.

By Lemma 3, each value computed in Step 4 is equal to $\max_{0 \le i < n} \delta_{P_i}(x,y)$. By Lemma 5, the largest of the values computed in Step 4 is equal to $\delta_C^{\max}$. This proves the correctness of the algorithm. To analyze the running time of the algorithm, it is clear that Step 1 takes $O(n)$ time. The closest-pair computation in Step 2 takes $O(n \log n)$ time; see [18]. In [15], it is shown that the size of the set S is $O(n)$. It is also shown there that if the points in the vertex set of C are stored in two lists X and Y, where the points in X are sorted by x-coordinates and the points in Y are sorted by y-coordinates, and if there are cross-pointers between these two lists, then the set S can be computed in $O(n)$ time. Therefore, Step 3 takes $O(n \log n)$ time. In Step 4, the algorithm

spends $O(1)$ time for each element of S. Since the size of S is $O(n)$, the total time for Step 4 is $O(n)$. Thus, the total time of the algorithm is $O(n \log n)$.

If the cycle C is a convex polygon, then we can improve the running time: In [14], it is shown that the closest pair can be computed in $O(n)$ time. Since C is a convex polygon, we can obtain the lists X and Y in $O(n)$ time. It follows that the entire algorithm runs in $O(n)$ time.

The following observations lead to an alternative $O(n)$–time algorithm for the case when C is a convex polygon. The Delaunay triangulation DT of the vertex set of C can be computed in $O(n)$ time, see [2]. This implies that the closest pair can be computed in the same time bound. We show that DT can be used to compute the set S in $O(n)$ time. A proof of the following lemma can be found in [6].

Lemma 6 *Consider the Delaunay triangulation DT of the vertex set of C, and let x and y be two distinct vertices that are not connected by an edge in DT. Then there exists a path $(x = x_0, x_1, x_2, \ldots, x_k = y)$ in DT, such that*

1. *for each i with $0 \leq i < k$, $|x_i x_{i+1}| < |xy|$ and*
2. *for each i with $0 \leq i \leq k$, $|xx_i| < |xy|$.*

Let (p, q) be the closest pair in the vertex set of C and let $R := 2|pq|$. Let DT' be the subgraph of DT consisting of all edges having length at most R. Lemma 6 implies that we obtain the set S (i.e., all pairs of points whose distance is at most R) by performing a breadth-first search from each vertex x of DT until we reach a vertex y such that $|xy| > R$. The total time for this is proportional to the size of S, which we know to be $O(n)$.

Theorem 2 *Given a polygonal cycle C on n vertices in the plane, we can compute $\delta_C^{\max}$ in $O(n \log n)$ time. If C is a convex polygon, $\delta_C^{\max}$ can be computed in $O(n)$ time.*

4 Approximating the Dilation of All Paths P_i

Consider again the polygonal cycle $C = (p_0, \ldots, p_{n-1}, p_0)$ whose vertices are points in the plane. Let $\epsilon > 0$ be a constant. In this section, we show that an approximation to the dilation of *each* path P_i $(0 \leq i < n)$, as well as an approximation to $\delta_C^{\min}$, can be computed in $O(n \log n)$ total time.

Our algorithm uses the *well-separated pair decomposition (WSPD)* of [3]. More specifically, we use the following lemma from [16], which states that the WSPD of the vertex set of any Euclidean graph G can be used to approximate the dilation of G. The statement of the lemma as we present it appears in Section 13.2.1 of [17].

Lemma 7 *Let V be a set of n points in the plane and let*

$$\{A_1, B_1\}, \{A_2, B_2\}, \ldots, \{A_m, B_m\}$$

be a WSPD for V with separation ratio $4(2 + \epsilon)/\epsilon$. For each j with $1 \leq j \leq m$, let a_j be an arbitrary point in A_j, and let b_j be an arbitrary point in B_j. For any connected Euclidean graph G with vertex set V, the following holds: For each j with $1 \leq j \leq m$, let $\delta_G(a_j, b_j)$ be the dilation between a_j and b_j in G, and let

$$t := \max_{1 \leq j \leq m} \delta_G(a_j, b_j).$$

Then

$$\delta_G/(1+\epsilon) \leq t \leq \delta_G,$$

where δ_G denotes the dilation of G.

Thus, in order to approximate the dilation of a Euclidean graph, it is sufficient to compute the dilation between $O(n)$ pairs of vertices. Moreover, the choice of these vertices depends only on the vertex set of the graph, it does not depend on the edges of the graph.

In a preprocessing step, we use the algorithm of [3] to compute, in $O(n \log n)$ time, a WSPD $\{A_j, B_j\}$, $1 \leq j \leq m = O(n)$, for the vertex set $\{p_0, \ldots, p_{n-1}\}$ of the cycle C, with separation ratio $4(2+\epsilon)/\epsilon$. For each j with $1 \leq j \leq m$, we pick an arbitrary point a_j in A_j, and an arbitrary point b_j in B_j. Our algorithm will compute, for each i with $0 \leq i < n$, the value

$$t_i := \max_{1 \leq j \leq m} \delta_{P_i}(a_j, b_j).$$

Observe that, by Lemma 7,

$$\delta_{P_i}/(1+\epsilon) \leq t_i \leq \delta_{P_i}.$$

Lemma 8 *Let $t^* := \min(t_0, t_1, \ldots, t_{n-1})$. Then*

$$\delta_C^{\min}/(1+\epsilon) \leq t^* \leq \delta_C^{\min}.$$

Proof. Let i be an index such that $t^* = t_i$ and let j be an index such that $\delta_C^{\min} = \delta_{P_j}$. Then

$$t^* \leq t_j \leq \delta_{P_j} = \delta_C^{\min}$$

and

$$\delta_C^{\min} = \delta_{P_j} \leq \delta_{P_i} \leq (1+\epsilon)t_i = (1+\epsilon)t^*.$$

□

We remark that, by a similar argument, $t^{**} := \max(t_0, t_1, \ldots, t_{n-1})$ can be shown to satisfy

$$\delta_C^{\max}/(1+\epsilon) \leq t^{**} \leq \delta_C^{\max}.$$

In other words, the algorithm that will be presented below can also be used to compute an approximation to $\delta_C^{\max}$ in $O(n \log n)$ time. We have seen in Section 3, however, that the exact value of $\delta_C^{\max}$ can be computed within the same time bound.

As mentioned above, our algorithm computes t_i for $i = 0, 1, \ldots, n-1$. The main idea is to maintain, for the current value of i, the m dilations $\delta_{P_i}(a_j, b_j)$ $(1 \leq j \leq m)$ in a balanced binary search tree T. Observe that, for any fixed index j, the value of $\delta_{P_i}(a_j, b_j)$ changes at most twice when i is increased from 0 to $n-1$. As a result, the total number of updates in T will be at most $2m$. We now present the details.

Let P denote the path $(p_0, p_1, \ldots, p_{n-1})$. Recall the relation $\prec_P$ of Section 2.1. We may assume without loss of generality that $a_j \prec_P b_j$ for each j with $1 \leq j \leq m$.

In the preprocessing step, we compute, in $O(n)$ time, the values $\Delta(p_i) = d_P(p_0, p_i)$ $(0 \leq i < n)$ and the total length L of the cycle C. Observe that, for $0 \leq i < n$, the distance $d_{P_i}(a_j, b_j)$ between a_j and b_j in the path P_i satisfies

$$d_{P_i}(a_j, b_j) = \begin{cases} \Delta(b_j) - \Delta(a_j) & \text{if } i = n-1 \text{ or } p_i \prec_P a_j \text{ or } b_j \prec_P p_{i+1}, \\ L - (\Delta(b_j) - \Delta(a_j)) & \text{otherwise.} \end{cases} \quad (1)$$

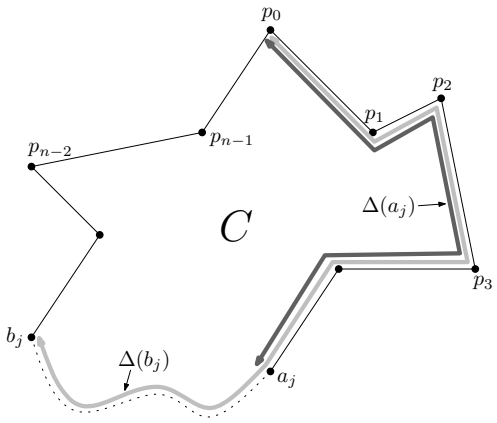

Figure 3: The cycle C.

For each j, there are exactly two paths between a_j and b_j on the cycle C. Therefore for $0 \leq i < n$, $d_{p_i}(a_j, b_j)$ can have at most two different values. The value changes when the deleted edge moves from one of the paths between a_j and b_j to the other path between them. So the value of $d_{P_i}(a_j, b_j)$ changes when $p_i = a_j$ or $p_i = b_j$. In the final part of the preprocessing step, we compute, for each i with $0 < i < n$, the set

$$S_i := \{j \mid a_j = p_i \text{ or } b_j = p_i\},$$

which contains all the indices j, $1 \leq j \leq m$, such that $d_{P_i}(a_j, b_j)$ is different from $d_{P_{i-1}}(a_j, b_j)$. Obviously, all these sets can be computed in $O(m) = O(n)$ time. After the preprocessing step, the algorithm proceeds as follows.

Step 1: Initialize an empty balanced binary search tree T (e.g., a red-black tree).

Step 2: For $j = 1, 2, \ldots, m$, use (1) to compute $d_{P_0}(a_j, b_j)$, compute $D_j := \delta_{P_0}(a_j, b_j)$ and insert D_j into T.

Step 3: Compute the maximum element t_0 in the tree T.

Step 4: For $i = 1, 2, \ldots, n-1$, perform the following Steps 4.1–4.2. (Observe that, at this moment, $D_j = \delta_{P_{i-1}}(a_j, b_j)$, $1 \leq j \leq m$.)

Step 4.1: For each element j in S_i, delete D_j from the tree T, use (1) to compute $d_{P_i}(a_j, b_j)$, compute the new value $D_j := \delta_{P_i}(a_j, b_j)$ and insert D_j into T.

Step 4.2: Compute the maximum element t_i in the tree T.

Step 5: Compute $t^* := \min_{0 \leq i < n} t_i$, and return the sequence $t_0, t_1, \ldots, t_{n-1}, t^*$.

The correctness of the algorithm follows from the discussion above. We have seen already that the preprocessing step takes $O(n \log n)$ time. Steps 1–3 take $O(m \log m) = O(n \log n)$ time. The total time for Step 4 is proportional to

$$\sum_{i=1}^{n-1} (|S_i| + 1) \log m \leq (2m + n) \log m = O(n \log n).$$

Theorem 3 *Given a polygonal cycle $C = (p_0, \ldots, p_{n-1}, p_0)$ on n vertices in the plane and a constant $\epsilon > 0$, in $O(n \log n)$ time, we can compute a sequence $t_0, \ldots, t_{n-1}, t^*$ of real numbers, such that*

$$\delta_{P_i}/(1+\epsilon) \leq t_i \leq \delta_{P_i}$$

for each $i = 0, 1, \ldots, n-1$ and

$$\delta_C^{\min}/(1+\epsilon) \leq t^* \leq \delta_C^{\min}.$$

5 Concluding Remarks

Recently, there has been a fair amount of work on the problem of computing the optimal dilation of a given (geometric) graph. In this paper we considered a variation of the problem where we are given a polygonal cycle and are supposed to choose one edge to remove such that the resulting polygonal path gives the smallest (or the largest) possible dilation. We presented an $O(n \log^3 n)$ expected time algorithm which given a polygonal cycle C, computes the edge whose removal makes minimum increase in the dilation of the generated path. In the case of maximizing the dilation we can compute the edge in $O(n \log n)$ time.

Acknowledgments

We would like to thank Jan Vahrenhold for fruitful discussions on the subject.

References

[1] P. K. Agarwal, R. Klein, C. Knauer, S. Langerman, P. Morin, M. Sharir, and M. Soss. Computing the detour and spanning ratio of paths, trees and cycles in 2d and 3d. *Discrete & Computational Geometry*. to appear.

[2] A. Aggarwal, L. J. Guibas, J. Saxe, and P. W. Shor. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete & Computational Geometry*, 4:591–604, 1989.

[3] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM*, 42:67–90, 1995.

[4] O. Cheong, H. Haverkort, and M. Lee. Computing a minimum-dilation spanning tree is NP-hard. In *CATS '07: Proceedings of the 19th Computing: The Australasian Theory Symposium*. CRPIT, 2007.

[5] K. L. Clarkson and P. W. Shor. Algorithms for diametral pairs and convex hulls that are optimal, randomized, and incremental. In *SCG '88: Proceedings of the fourth annual symposium on Computational geometry*, pages 12–17, New York, NY, USA, 1988. ACM Press.

[6] M. T. Dickerson, R. L. Drysdale, and J. R. Sack. Simple algorithms for enumerating interpoint distances and finding k nearest neighbors. *International Journal of Computational Geometry & Applications*, 2:221–239, 1992.

[7] D. Eppstein and K. A. Wortman. Minimum dilation stars. In *SCG '05: Proceedings of the 21st Annual ACM Symposium on Computational Geometry*, pages 321–326, New York, NY, USA, 2005. ACM Press.

[8] M. Farshi, P. Giannopoulos, and J. Gudmundsson. Finding the best shortcut in a geometric network. In *SCG '05: Proceedings of the 21st Annual ACM Symposium on Computational Geometry*, pages 327–335, New York, NY, USA, 2005. ACM Press.

[9] S. J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[10] M. L. Fredman, J. Komlos, and E. Szemeredi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31:538–544, 1984.

[11] P. Giannopoulos, C. Knauer, and D. Marx. Minimum-dilation tour (and path) is NP-hard. In *EWCG '07: Proceedings of the 23rd European Workshop on Computational Geometry*, pages 18–21, 2007.

[12] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12:28–35, 1983.

[13] R. Klein and M. Kutz. Computing geometric minimum-dilation graphs is np-hard. In Michael Kaufmann and Dorothea Wagner, editors, *Graph Drawing, Karlsruhe, Germany, September 18-20, 2006*, pages 196–207. Springer, 2007.

[14] D. T. Lee and F. P. Preparata. The all nearest-neighbor problem for convex polygons. *Information Processing Letters*, 7:189–192, 1978.

[15] H. P. Lenhof and M. Smid. Sequential and parallel algorithms for the k closest pairs problem. *International Journal of Computational Geometry & Applications*, 5:273–288, 1995.

[16] G. Narasimhan and M. Smid. Approximating the stretch factor of Euclidean graphs. *SIAM Journal on Computing*, 30:978–989, 2000.

[17] G. Narasimhan and M. Smid. *Geometric Spanner Networks*. Cambridge University Press, Cambridge, UK, 2007.

[18] M. Smid. Closest-point problems in computational geometry. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 877–935. Elsevier Science, Amsterdam, 2000.