Dilation-Optimal Edge Deletion in Polygonal Cycles*

Michiel Smid[4,***], and Yajun Wang[5]

[1] Department of Computer Science and Engineering, POSTECH, Pohang, Korea
heekap@postech.ac.kr
[2] Department of Mathematics and Computing Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
m.farshi@tue.nl
[3] Institut für Informatik, Freie Universität Berlin, Takustraße 9, D–14195 Berlin, Germany
christian.knauer@inf.fu-berlin.de
[4] School of Computer Science, Carleton University, Ottawa, Ontario, K1S 5B6, Canada. michiel@scs.carleton.ca
[5] Department of Computer Science and Engineering, HKUST, Hong Kong S.A.R, China
yalding@cse.ust.hk

Abstract. Let C be a polygonal cycle 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

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. We consider a

* Part of this work was done during the Korean Workshop on Computational Geometry at Schloß Dagstuhl in 2006.

** This author was supported by Ministry of Science, Research and Technology of I. R. Iran.

*** This author was supported by NSERC.

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. We measure the quality of the resulting polygonal path P by its *dilation* (or stretch factor) δ_P.

Recall that the dilation between two distinct *vertices* x and y of the path P is defined as $\delta_P(x,y) := d_P(x,y)/|xy|$, where $d_P(x,y)$ denotes the Euclidean length of the subpath of P connecting x and y, and $|xy|$ denotes the Euclidean distance between x and y. For convenience we define $\delta_P(x,x) := 1$. The dilation between two *sets* X and Y of *vertices* of P is defined as

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

the dilation of a *set* X of *vertices* of P is defined as $\delta_P(X) := \delta_P(X,X)$, and the *dilation* of the path P is defined as

$$\delta_P := \delta_P(P) = \max\{\delta_P(x,y) \mid x \text{ and } y \text{ are vertices of } P\}.$$

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 $0 \le i < n$) 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 $\delta_C^{\min} := \min_{0\le i<n} \delta_{P_i}$ and $\delta_C^{\max} := \max_{0\le i<n} \delta_{P_i}$. We will prove the following results :

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.*

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.*

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 $\delta_{P_i}/(1+\epsilon) \le t_i \le \delta_{P_i}$ for each $i = 0,1,\ldots,n-1$ and $\delta_C^{\min}/(1+\epsilon) \le t^* \le \delta_C^{\min}$.*

In Sect. 2, we prove Theorem 1. We start in Sect. 2.1 by describing an approach of [1] to estimate the dilation of a polygonal path; see also [7]. These ideas will play a central role in the algorithm we give in Sect. 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 Sect. 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 Sect. 2.2. Thus, this reduction incurs a logarithmic slowdown of the decision procedure.

In Sect. 3, we prove Theorem 2. 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 $\delta_C^{\max}$.

Theorem 3 is proved in Sect. 4. The algorithm uses the well-separated pair decomposition of [2] and a result of [10], 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 $\delta_{P_0}(x, y), \dots, \delta_{P_{n-1}}(x, y)$ 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

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 $\kappa > 1$; see also [7]. We describe those parts of this algorithm which are relevant for us.

Let $P = (p_0, \dots, p_{n-1})$ be a polygonal path whose n vertices are points in the plane and let $\kappa \geq 1$ be a real number. 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

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

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 $p <_P q$. We then get the following lemma.

Lemma 1. *For any two vertices p and q of P with $p <_P q$, 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

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

□

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 $X <_P Y$. 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.

Lemma 2. *For any two subsets X and Y of the vertex set of P with $X <_P Y$, we have $\delta_P(X,Y) < \kappa$ if and only if $h_P(Y)$ 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 $X <_P Y$: The Voronoi diagram $V_P(X)$ can be computed in $O(|X| \log |X|)$ time, c.f. [4]. 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. [6]. 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.

If $R = (r_1, \ldots r_m)$ and $Q = (q_1, \ldots, q_n)$ are two polygonal paths having the property that $l(R) = r_m = q_1 = f(Q)$, then we denote the concatenation of R and Q by $R \oplus Q$. Thus, $R \oplus Q$ is the polygonal path $(r_1, \ldots, r_m, q_2, \ldots, q_n)$.

In order to facilitate a recursive approach, we will consider the following more general problem: Assume that (the edge 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, whether or not the dilation of $C \setminus \{e\}$ is less than κ. If we take $T = \{p\}$, where p is an arbitrary vertex of C, then we obtain the original problem.

The details of the decision algorithm is presented in Algorithm 2.1. The correctness of Algorithm 2.1 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 edges on B. The expected running time $t(n)$ of the algorithm can therefore be written as

Algorithm 2.1. Decision-Algorithm

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

$t(n) = O(n \log^2 n) + r(n)$ 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 1 illustrates the recursion tree of Algorithm 2.1. The nodes of the tree are labeled according to a BFS-numbering where the first (left) child of a node corresponds to the recursive call in Step 11 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 $O(\log n)$ bottom paths.

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 $l(B'_i) = f(B'_{i+1})$ for $1 \leq i < k$. Thus, we have $T' = B'_1 \oplus \ldots \oplus B'_k$. Given a new polygonal chain B' with $f(B') = l(T')$, 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

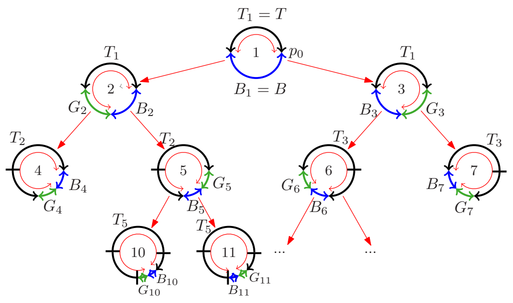

Fig. 1. The recursion tree. Note that $G_{2i} := B_{2i+1}$ and $G_{2i+1} := B_{2i}$.

of [1], we can decide in $O(|B'| \log |B'|)$ 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 $B'_i <_{T' \oplus B'} B'$, 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

$$\omega_{T' \oplus B'}(t) = \omega_{B'_i}(t) + \ell_{i-1} \qquad \text{and} \qquad \omega_{T' \oplus B'}(b') = \omega_{B'}(b') + \ell_k.$$

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 2.1 runs in $O(n \log^3 n)$ time.

A Faster Implementation. In the i-th step of the recursion, 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 [5] 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 [5], they assume that the entries come from a universe set and the memory is able to randomly access each entry in constant time. Recall that the construction of [5] 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 2.1 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 [3] used a similar randomized approach to compute 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 2.1 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 2.1 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 2.1, 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 $\delta_{P_1}, \dots, \delta_{P_n}$ 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 $O(n \log^3 n)$. This completes the proof of Theorem 1.

3 Dilation-Maximal Edge Deletion in a Polygonal Cycle

In this section, we will prove Theorem 2. That is, 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 $(p_0, \dots, p_i)$ 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 $\frac{L}{|pq|} \le 2 \cdot \max_{0 \le i < n} \delta_{P_i}(p, q)$. $\square$

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 $x \neq y$ and $|xy| \leq 2|pq|$. The following lemma states that it suffices to consider the elements of S to compute $\delta_C^{\max}$.

Lemma 5. *We have* $\delta_C^{\max} = \max_{(x,y)\in S} \ \max_{0\leq i<n} \delta_{P_i}(x, y)$.

Proof. It is clear that

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

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., $|ab| \leq 2|pq|$), then the proof is complete.

By Lemma 3, we have the following inequality, which follows that

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

This implies that $|ab| \leq 2|pq|$. □

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 $\Delta(p_i)$ $(0 \leq i < n)$ 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 $x \neq y$ and $|xy| \leq 2|pq|$.
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 $\max_{0\leq i<n} \delta_{P_i}(x, y)$. 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 [12]. In [9], 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 [8], 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.

4 Approximating the Dilation of All Paths P_i

Consider again the polygonal cycle $C = (p_0, \dots, p_{n-1}, p_0)$ whose vertices are points in the plane. Let $\epsilon > 0$ be a constant. In this section, we prove Theorem 3. That is, 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 [2]. More specifically, we use the following lemma from [10], 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 [11].

Lemma 6. *Let V be a set of n points in the plane and let $\{A_1, B_1\}, \{A_2, B_2\}, \dots, \{A_m, B_m\}$ 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 [2] 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 $\{p_0, \dots, p_{n-1}\}$ 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 6, $\delta_{P_i}/(1+\epsilon) \leq t_i \leq \delta_{P_i}$.

Lemma 7. *Let $t^* := \min(t_0, t_1, \dots, t_{n-1})$. 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 $\delta_C^{\min} = \delta_{P_j} \leq \delta_{P_i} \leq (1+\epsilon)t_i = (1+\epsilon)t^*$. $\square$

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 Sect. 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 $<_P$ of Sect. 2.1. We may assume without loss of generality that $a_j <_P b_j$ 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

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

In the final part of the preprocessing step, we compute, for each i with $0 < i < n$, the set $S_i := \{j \mid 1 \leq j \leq m, a_j = p_i \text{ or } b_j = p_i\}$. 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 it 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 it 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

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

This completes the proof of Theorem 3.

5 Concluding Remarks and Acknowledgments

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

References

[1] Agarwal, P.K., Klein, R., Knauer, C., Sharir, M.: Computing the detour of polygonal curves. Technical Report B 02-03, Fachbereich Mathematik und Informatik, Freie Universität Berlin (2002)

[2] Callahan, P.B., Kosaraju, S.R.: 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)

[3] Clarkson, K.L., Shor, P.W.: Algorithms for diametral pairs and convex hulls that are optimal, randomized, and incremental. In: SCG 1988: Proceedings of the fourth annual symposium on Computational geometry, pp. 12–17. ACM Press, New York (1988)

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

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

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

[7] Langerman, S., Morin, P., Soss, M.: Computing the maximum detour and spanning ratio of planar paths, trees and cycles. In: Alt, H., Ferreira, A. (eds.) STACS 2002. LNCS, vol. 2285, pp. 250–261. Springer, Heidelberg (2002)

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

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

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

[11] Narasimhan, G., Smid, M.: Geometric Spanner Networks. Cambridge University Press, Cambridge (2007)

[12] Smid, M.: Closest-point problems in computational geometry. In: Sack, J.-R., Urrutia, J. (eds.) Handbook of Computational Geometry, pp. 877–935. Elsevier Science, Amsterdam (2000)