An External Memory Data Structure for Shortest Path Queries (Extended Abstract)*

David Hutchinson[1,**], Anil Maheshwari[1,**], and Norbert Zeh[1,2,***]

[1] School of Computer Science, Carleton University, Ottawa, Canada
[2] Fakultät für Math. und Inf., Friedrich-Schiller-Universität Jena, Germany
{hutchins,maheshwa,nzeh}@scs.carleton.ca

Abstract. We present results related to satisfying shortest path queries on a planar graph stored in external memory. In particular, we show how to store rooted trees in external memory so that bottom-up paths can be traversed I/O-efficiently, and we present I/O-efficient algorithms for triangulating planar graphs and computing small separators of such graphs. Using these techniques, we can construct a data structure that allows for answering shortest path queries on a planar graph I/O-efficiently.

1 Introduction

Motivation. Answering shortest path queries in graphs is an important and well studied problem. Applications include communication systems, transportation problems, scheduling, computation of network flows, and geographic information systems (GIS). Typically, an underlying geometric structure is represented by an equivalent combinatorial structure, which is often a weighted, planar graph.

Model of Computation. In many applications data sets are too large to fit into the main memory of existing machines. In such cases, conventional internal memory algorithms can be inefficient, accessing their data in a random fashion, and causing many data transfers between internal and external memory. This I/O-bottleneck is becoming more significant as parallel computing gains popularity and CPU speeds increase, since disk speeds are not keeping pace. Several computational models for estimating the I/O-efficiency of algorithms have been developed [10, 11, 3]. We adopt the *parallel disk model* PDM [10] as our model of computation for this paper due to its simplicity, and the fact that we consider only a single processor.

In the PDM, an *external memory*, consisting of D disks, is attached to a machine with memory size M data items. Each of the disks is divided into blocks of B consecutive data items. Up to D blocks, at most one per disk, can be transferred between internal and external memory in a single I/O operation. The complexity of an algorithm is the number of I/O operations it performs.

* For details see [12].
** Research partially supported by NSERC.
*** Research partially supported by Studienstiftung des deutschen Volkes.

Previous Results. Frederickson [5] proposed an $O\left(N\sqrt{\log N}\right)$ time algorithm to compute shortest paths in planar graphs using separators. This technique was extended by Djidjev [4] who developed an $O(S)$-space data structure ($N \leq S \leq N^2$) that answers distance queries on planar graphs in $O(N^2/S)$ time in internal memory. The corresponding shortest path can be reported in time proportional to the length of the reported path.

Lipton and Tarjan [6] presented a linear-time algorithm for finding a $\frac{2}{3}$-separator of size $O\left(\sqrt{N}\right)$ for any planar graph.

In the PDM, sorting an array of size N takes $sort(N) = \Theta\left(\frac{N}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right)$ I/Os [10, 9]. Scanning an array of size N takes $scan(N) = \Theta\left(\frac{N}{DB}\right)$ I/Os. For a comprehensive survey of external memory algorithms, refer to [9]. The only external memory shortest path algorithm known to us is the single source shortest path algorithm by Crauser *et al.* [2], which takes $O\left(\frac{|V|}{D} + \frac{|E|}{DB}\log_{\frac{M}{B}}\frac{|E|}{B}\right)$ I/Os with high probability on a random graph with random weights. We do not know of previous work on computing separators in external memory. One can use the PRAM simulation results of Chiang *et al.* [1] together with known PRAM separator algorithms. Unfortunately, the PRAM simulation introduces $O(sort(N))$ I/Os for every PRAM step, and so the resulting I/O complexity is not attractive for our purposes.

Our Results. The main results of this paper are listed below. Details can be found in [12].

1. In Sect. 3, we present a blocking to store a rooted tree T of size N in at most $\left(2 + \frac{2}{1-\tau}\right)\frac{N}{B} + D$ blocks so that a path of length K *towards the root* can be traversed in at most $\left\lceil\frac{K}{\tau DB}\right\rceil + 1$ I/Os, for $0 < \tau < 1$. For fixed τ, the tree uses optimal $O(|T|/B)$ space and traversing a path takes optimal $O(K/DB)$ I/Os. Using the best previous result by Nodine *et al.* [8], the tree would use the same amount of space within a constant factor, but traversing a path would take $O(K/\log_d(DB))$ I/Os, where d is the maximal degree of the vertices in the tree.
2. In Sect. 4, we present an external memory algorithm to compute a separator of size $O\left(\sqrt{N}\right)$ for an embedded planar graph in $O(sort(N))$ I/Os, provided that a breadth-first search tree (BFS-tree) of the graph is given. Our algorithm is based on the planar separator technique in [6]. The main challenge in designing an external memory algorithm for this problem is to determine a good separator corresponding to a fundamental cycle.
3. In Sect. 5, we describe an external memory algorithm which triangulates an embedded planar graph in $O(sort(N))$ I/O operations.
4. Results 1-3, above, are the main techniques that we use to construct an external memory data structure for answering shortest path queries online. Our data structure uses $O\left(N^{3/2}/B\right)$ blocks of external storage and answers online distance and shortest path queries in $O\left(\sqrt{N}/DB\right)$ and $O\left(\left(\sqrt{N} + K\right)/DB\right)$ I/Os, respectively, where K is the number of vertices on the path.

The separator and triangulation algorithms may be of independent interest, since graph separators are used in the design of efficient divide-and-conquer graph algorithms and many graph algorithms assume triangulated input graphs.

2 Preliminaries

A *graph* $G = (V, E)$ is a pair of sets V and E, where V is called the *vertex set* and E is called the *edge set* of G. Each edge in E is an unordered pair $\{v, w\}$ of vertices v and w in V. A graph G is called *planar* if it can be drawn in the plane so that no two edges intersect, except possibly at their endpoints. Such a drawing defines, for each vertex v of G, an order of the edges incident to v clockwise around v. We call G *embedded* if we are given this order for every vertex of G. By Euler's formula, $|E| \leq 3|V| - 6$ for planar graphs. A *path* from a vertex v to a vertex w in G is a list $p = \langle v = v_0, \ldots, v_k = w \rangle$ of vertices, where $\{v_i, v_{i+1}\} \in E$ for $0 \leq i < k$. A graph G is *connected* if there is a path between any two vertices in G. A *subgraph* $G' = (V', E')$ of G is a graph with $V' \subseteq V$ and $E' \subseteq E$. *Connected components* of G are the maximal connected subgraphs of G. Let $c : E \to \mathbb{R}^+$ be a mapping that assigns non-negative costs to the edges of G. The *cost* of a path $p = \langle v_0, \ldots, v_k \rangle$ is defined as $|p| = \sum_{i=0}^{k-1} c(\{v_i, v_{i+1}\})$. A shortest path $\pi(v, w)$ is a path from v to w of minimal cost. Let $w : V \to \mathbb{R}^+$ be a mapping that assigns non-negative weights to the vertices of G such that $\sum_{v \in V} w(v) \leq 1$. The *weight* of a subgraph H of G is the sum of the weights of the vertices in H. An *ϵ-separator*, $0 < \epsilon < 1$, of G is a subset C of V whose removal partitions G into two subgraphs, A and B, each of weight at most ϵ, so that there is no edge in G that connects any vertex in A to any vertex in B. We will describe results on paths in a tree which originate at an arbitrary node of the tree and proceed to the root. We will refer to such paths as *bottom-up* paths.

3 Blocking Rooted Trees

In this section we describe a blocking of a rooted tree T so that we can traverse a bottom-up path from a given vertex v of T in an I/O-efficient manner. We assume that all accesses to T are read-only. Thus, we can store each vertex an arbitrary number of times and use redundancy to reduce the number of blocks that have to be read. However, this increases the space requirements. The following theorem gives a trade-off between the space requirements of the data structure and the I/O-efficiency of the tree traversal. The proof follows from Lemmas 1 and 2. The proof of Lemma 2 and an algorithm to construct such a blocking for a given tree in $O(sort(N))$ I/Os are given in [12].

Theorem 1. *Given a rooted tree T of size N and a constant τ, $0 < \tau < 1$, we can store T in at most $\left(2 + \frac{2}{1-\tau}\right)\frac{N}{B} + D$ blocks on D parallel disks so that traversing any bottom-up path of length K in T takes at most $\left\lceil \frac{K}{\tau DB} \right\rceil + 1$ I/Os.*

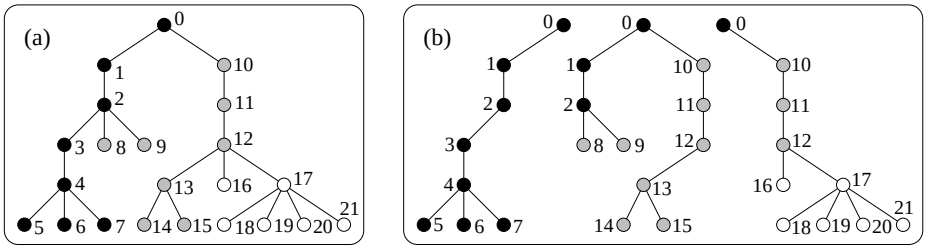

Fig. 1. (a) A rooted tree T_i with its vertices labelled with their preorder numbers. Assuming that $t = 8$, V_0, V_1, and V_2 are the sets of black, grey, and white vertices, respectively. (b) The subtrees $T_i(V_1)$, $T_i(V_2)$, and $T_i(V_3)$ from left to right.

Intuitively, our approach is as follows. We cut T into layers of height τDB. This divides every bottom-up path of length K in T into subpaths of length τDB; each subpath stays in a particular layer. We ensure that each such subpath is stored in a single block and can thus be traversed at the cost of a single I/O operation. This gives us the desired I/O-bound because any path of length K is divided into at most $\lceil \frac{K}{\tau DB} \rceil + 1$ subpaths.

More precisely, let $h(T)$ represent the height of T, and let $h' = \tau DB$ be the height of the layers to be created (we assume that h' is an integer). Let the level of a vertex v be the number of edges in the path from v to the root r of T. Cut T into layers $L_0, \ldots, L_{\lceil h(T)/h' \rceil - 1}$, where layer L_i is the subgraph of T induced by the vertices on levels ih' through $(i+1)h' - 1$. Each layer is a forest of rooted trees whose heights are at most h'. Suppose that there are r such trees, taken over all layers. Let $T_1, \ldots, T_r$ denote these trees for all layers of T.

Lemma 1. *Given a rooted tree T_i of height at most τDB, we can divide T_i into subtrees $T_{i,0}, \ldots, T_{i,s}$ with the following properties: (1) $|T_{i,j}| \leq DB$, for all $0 \leq j \leq s$, (2) $\sum_{j=0}^{s} |T_{i,j}| \leq \left(1 + \frac{1}{1-\tau}\right) |T_i|$, and (3) for every leaf l of T_i, there is a subtree $T_{i,j}$ containing the whole path from l to the root of T_i.*

Proof sketch. If $|T_i| \leq DB$, we "divide" T_i into one subtree $T_{i,0} = T_i$. Then Properties 1–3 trivially hold. So assume that $|T_i| > DB$. Given a preorder numbering of the vertices of T_i, let v_k be the vertex with preorder number k. Let $h' = \tau DB$, $t = DB - h'$, and $s = \lceil |T_i|/t \rceil - 1$. We define vertex sets $V_0, \ldots, V_s$, where $V_j = \{v_{jt}, \ldots, v_{(j+1)t-1}\}$ for $0 \leq j \leq s$ (see Fig. 1(a)). The subtree $T_{i,j} = T_i(V_j)$ is the subtree of T_i consisting of all vertices in V_j and their ancestors in T_i (see Fig. 1(b)). We claim that these trees $T_{i,j}$ have Properties 1–3.

Property 3 is ensured by including the ancestors in T_i of all vertices in V_j in $T_i(V_j)$. Property 2 follows, if we can prove Property 1 because then $\sum_{j=0}^{s} |T_{i,j}| \leq \sum_{j=0}^{s} DB = (s+1)DB \leq \left(\frac{1}{1-\tau} + 1\right) |T_i|$.

It can be shown that every vertex in $T_i(V_j)$ that is not in V_j is an ancestor of v_{jt}. As the height of T_i is at most h', there can be at most h' such ancestors of v_{jt}. Moreover, $|V_j| \leq DB - h'$. Thus, $|T_i(V_j)| \leq DB$. □

Lemma 2. *If a rooted tree T of size N is partitioned into subtrees $T_{i,j}$ such that the properties 1–3 in Lemma 1 hold, then T can be stored using $\left(2+\frac{2}{1-\tau}\right)\frac{N}{B}+D$ blocks of external memory, and any bottom-up path of length K in T can be traversed in at most $\left\lceil\frac{K}{\tau DB}\right\rceil+1$ I/Os.*

4 Separating Embedded Planar Graphs

We now present an external memory algorithm for separating embedded planar graphs. It is based on Lipton and Tarjan's [6] linear-time separator algorithm. The input to our algorithm is an embedded planar graph G and a spanning forest F of G. Every tree in F is a rooted BFS-tree of the respective connected component. The graph G is represented by its vertex set V and its edge set E. To represent the embedding, let the edges incident to a vertex v be numbered in counterclockwise order around v. This defines two numbers $n_v(e)$ and $n_w(e)$, stored with every edge $e=\{v,w\}$. The spanning forest F is given implicitly by marking every edge of G as tree or non-tree edge and storing, with each vertex v in V, the name of its parent $p(v)$ in F. We prove the following theorem.

Theorem 2. *Given an embedded planar graph G with N vertices and a BFS-tree[1] T of G, a $\frac{2}{3}$-separator of G of size at most $2\sqrt{2}\sqrt{N}$ can be computed in $O(sort(N))$ I/Os.*

Proof sketch. W.l.o.g., we assume that the given graph is connected. If it is not, we can compute its connected components in $O(sort(N))$ I/Os [1] and compute a separator of the component with weight greater than $\frac{2}{3}$, if any. Moreover we assume that G is triangulated. If it is not, it can be triangulated in $O(sort(N))$ I/Os using the algorithm in Sect. 5.

The separator consists of two parts. First we compute two levels l_0 and l_2 in G's BFS-tree T whose removal divides G into three parts G_1, G_2, G_3 with $|G_1|, |G_3| \leq \frac{2}{3}$ and such that $L(l_0)+L(l_2) \leq 2\sqrt{2}\sqrt{N}-2(l_2-l_0-1)$, where $L(l)$ is the number of vertices on level l. Lipton and Tarjan [6] proved that such levels l_0 and l_2 exist. Computing levels l_0 and l_2 takes $O(sort(N))$ I/Os using a generalization of the list ranking algorithm in [1], sorting, and scanning.

To separate G_2 into components of weights at most $\frac{2}{3}$ each, we shrink levels 0 through l_0 to a single vertex, remove levels l_2 and below, and retriangulate the resulting graph. Call the resulting graph G'. We construct G', a spanning tree T' of G', and an embedding of G' in $O(sort(N))$ I/Os using sorting, scanning, and the triangulation algorithm in Sect. 5. The separator of G_2 is a simple cycle separator of G', which contributes at most $2(l_2-l_0-1)$ vertices to the separator because the height of T' is l_2-l_0. Thus, the total size of the separator is at most $2\sqrt{2}\sqrt{N}$. Lemma 3 states that a simple cycle separator of G' can be computed in $O(sort(N))$ I/Os. □

[1] The currently best known BFS-algorithm [7] takes $O\left(|V|+\frac{|E|}{|V|}sort(|V|)\right)$ I/Os.

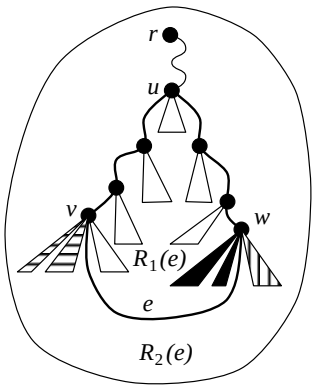

Fig. 2. A non-tree edge e and its fundamental cycle $c(e)$ shown in bold. $R_1(e)$ is the set of vertices embedded inside the cycle and $R_2(e)$ is the set of vertices embedded outside the cycle.

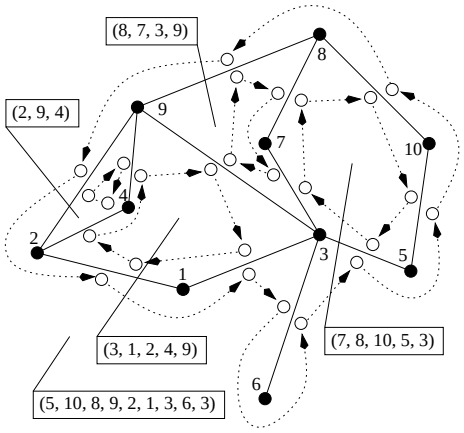

Fig. 3. A given graph G (black vertices and solid lines). White vertices and dotted arrows represent the graph $\hat{G}$ for G. Every face f of $\hat{G}$ is labelled with its corresponding vertex list F_f.

Finding a Small Simple Cycle Separator. Every non-tree edge $e = \{v, w\}$ in G' defines a fundamental cycle $c(e)$ consisting of e itself and the two paths in the tree T' from the vertices v and w to the lowest common ancestor (LCA) u of v and w (see Fig. 2). Any fundamental cycle $c(e)$ separates G' into two subgraphs $R_1(e)$ and $R_2(e)$, one induced by the vertices embedded inside $c(e)$ and the other induced by those embedded outside. Lipton and Tarjan showed that there is a non-tree edge e in G' such that $R_1(e)$ and $R_2(e)$ have weights at most $\frac{2}{3}$ each.

Lemma 3. *Given a triangulated graph G' with N vertices and a BFS-tree T' of G'. A $\frac{2}{3}$-simple cycle separator of size at most $2h(T') - 1$ for G' can be computed in $O(sort(N))$ I/Os.*

Proof sketch. For every vertex v in T', we compute a 4-tuple $A(v) = (n(v), \nu(v), \delta(v), W(v))$ of the following labels: (1) its preorder number $n(v)$, (2) its "weighted preorder number" $\nu(v) = \sum_{n(u) \leq n(v)} w(u)$, (3) the total weight $\delta(v)$ of v and all its ancestors in T', and (4) the weight $W(v)$ of all vertices in the subtree of T' rooted at v. It is essential that the preorder numbers $n(v)$ and $\nu(v)$ respect the embedding. That is, if $p(v)$, w_1, and w_2 appear in counterclockwise order around vertex v, then w_1 has a smaller preorder number than w_2. Less formally, this means that the subtrees in T' are labelled in left-to-right order. In [12] it is shown how to compute these labels in $O(sort(N))$ I/Os using known external memory techniques such as sorting, scanning, and time-forward processing [1].

We store the tuples $A(v), A(w)$, and $A(u)$ with every edge $e = \{v, w\}$, where u is the LCA of v and w. Computing all LCAs takes $O(sort(N))$ I/Os [1]. To

copy the appropriate vertex labels to all edges, we have to sort and scan the vertex and edge sets of G'.

For every vertex v of T', let $e_0, \ldots, e_k$ be the set of edges incident to it in counterclockwise order, and let $e_0 = \{v, p(v)\}$. We define $t(e_i) = 0$, if e_i is a non-tree edge, and $t(e_i) = W(w_i)$, if $e_i = \{v, w_i\}$ is a tree edge and $v = p(w_i)$. We compute labels $t_v(e_i) = t_v(e_{i-1}) + t(e_i)$, for $0 < i \leq k$, and $t_v(e_0) = 0$, sorting and scanning the edge set of G'.

Given a non-tree edge e, the weights of $R_1(e)$ and $R_2(e)$ can now be computed from the labels stored locally with e: Consider Fig. 2. A non-tree edge $e = \{v, w\}$ is shown and u is the LCA of v and w in T'. Four classes of subtrees are indicated by different patterns in Fig. 2. The vertices in the vertically hatched subtree are not important to the algorithm, but are included for completeness. The set of vertices $R_1(e)$, embedded inside the cycle, are the vertices in the white and black subtrees. The vertices in the horizontally hatched and white subtrees and on the tree path from u to w are exactly the vertices with preorder numbers between $n(v)$ and $n(w)$. Thus, their total weight is $\nu(w) - \nu(v)$. The total weight of the vertices in the horizontally hatched trees is $t_v(e)$; the total weight of the vertices on the path from u to w is $\delta(w) - \delta(u)$; the total weight of the black trees is $t_w(e)$. Thus, the total weight of vertices in $R_1(e)$ is $\nu(w) - \nu(v) - t_v(e) + t_w(e) - \delta(w) + \delta(u)$. The total weight of the vertices on $c(e)$ is $\delta(v) + \delta(w) - 2\delta(u) + w(u)$. Hence, the total weight of vertices in $R_2(e)$ is $w(G') - w(R_1(e)) - \delta(v) - \delta(w) + 2\delta(u) - w(u)$.

Thus, we can scan the edge set of G' and stop at the first non-tree edge with $w(R_1(e)), w(R_2(e)) \leq \frac{2}{3}$. The fundamental cycle $c(e)$ can be reported in $O(sort(N))$ I/Os by sorting the vertex set of G' by levels and preorder numbers in T' and then reporting on every level lower than that of u, those vertices with greatest preorder number less than $n(v)$ and $n(w)$, respectively. □

5 Triangulating Embedded Planar Graphs

In this section we present an $O(sort(N))$-algorithm to triangulate a connected embedded planar graph $G = (V, E)$. We assume the same representation of G and its embedding as in the previous section. Our algorithm consists of two phases. First we identify the faces of G. We represent each face f by a list of vertices on its boundary, sorted clockwise around the face. In the second phase, we use this information to triangulate the faces of G. The following theorem follows from Lemmas 4 and 6 below.

Theorem 3. *An embedded planar graph can be triangulated in $O(sort(N))$ I/Os.*

Identifying Faces. We compute a list F which is the concatenation of vertex lists F_f, one for each face of G. For a given face f, the list F_f is the clockwise sequence of vertices around face f. The list F_f may contain more than one copy of the same vertex, depending on how often this vertex is visited in a clockwise traversal of the face boundary.

TriangulateFaces(G, F):

1: Make all faces of G simple:
For each face f, (a) mark the first appearance of each vertex v in F_f, (b) append a marked copy of the first vertex in F_f to the end of F_f, and (c) scan F_f backward and remove each unmarked vertex v from f and F_f by adding a chord between its predecessor and successor in the current list.

2: Triangulate the simple faces:
Let $F_{\hat{f}} = \langle v_0, \ldots, v_k \rangle$. Add "temporary chords" $\{v_0, v_i\}$, $2 \leq i \leq k-1$, to $\hat{f}$.

3: Mark conflicting chords:
Sort E lexicographically, ensuring that edge $\{v, w\}$ is stored before all "temporary chords" $\{v, w\}$. Scan E and mark all occurrences of each edge, except the first, as "conflicting". Restore the original order of all edges and "temporary chords".

4: Retriangulate conflicting faces:
For each face $\hat{f}$, let $D_{\hat{f}} = \langle \{v_0, v_2\}, \ldots, \{v_0, v_{k-1}\} \rangle$ be the list of "temporary chords". Scan $D_{\hat{f}}$ until we find the first conflicting chord $\{v_0, v_i\}$. Replace $\{v_0, v_i\}, \ldots, \{v_0, v_{k-1}\}$ by chords $\{v_{i-1}, v_{i+1}\}, \ldots, \{v_{i-1}, v_k\}$.

Algorithm 1: Triangulating the faces of G.

Lemma 4. *The list F can be constructed in $O(sort(N))$ I/Os.*

Proof sketch. We first compute a graph $\hat{G}$ which is comprised of disjoint directed cycles. Each cycle represents a clockwise traversal of the boundary of a face f of G. Given a cycle in $\hat{G}$ that represents a face f of G, every vertex in this cycle represents an edge on the boundary of f. We construct F_f as the list of first endpoints of these edges in clockwise order around f (see Fig. 3).

Two vertices in $\hat{G}$ that are consecutive on a cycle of $\hat{G}$ represent two edges that are consecutive on the boundary of a face of G in clockwise order. Thus, these two edges are consecutive around a vertex of G in counterclockwise order. The graph $\hat{G}$ contains vertices $v_{(v,w)}$ and $v_{(w,v)}$ for every edge $\{v, w\}$ of G and edges $(v_{(u,v)}, v_{(v,w)})$, where $\{u, v\}$ and $\{w, v\}$ are consecutive counterclockwise around v. These vertex and edge sets can be computed sorting and scanning the vertex and edge sets of G.

We identify the cycles of $\hat{G}$ as its connected components using the algorithm in [1], sort and scan the edge set of $\hat{G}$ to remove an arbitrary edge from every such cycle. This transforms every cycle into a list, which can be ranked and then sorted by rank. As a result, the vertices of $\hat{G}$ are sorted clockwise around the faces of G. We scan these sorted lists to construct F. As we only use scanning, sorting, and the list ranking technique in [1], the complexity of this algorithm is $O(sort(N))$. □

Triangulating Faces. We triangulate each face f in four steps (see Algorithm 1). First, we reduce f to a simple face $\hat{f}$. (A face is simple if each vertex on its boundary is visited only once in a clockwise traversal of the boundary.) This reduces the list F_f to $F_{\hat{f}}$. In the second step, we triangulate $\hat{f}$. We ensure that there are no multiple edges in $\hat{f}$, but we might add *conflicting* edges (edges with

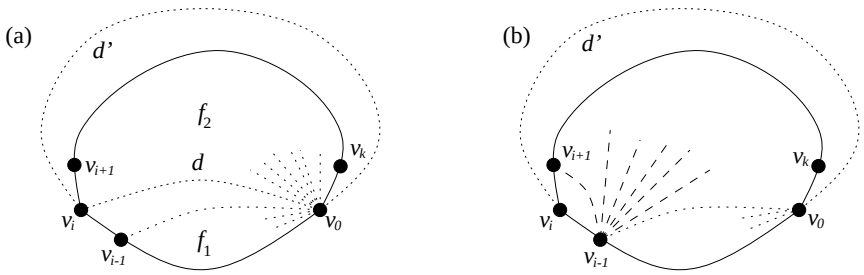

Fig. 4. (a) A simple face $\hat{f}$. Chord d conflicts with d' and divides $\hat{f}$ into two parts f_1 and f_2. One of them, f_1, is conflict-free. Vertex v_{i-1} is the third vertex of the triangle in f_1 that has d on its boundary. (b) The conflict-free triangulation of $\hat{f}$.

the same endpoints) to adjacent faces. (See Fig. 4 for an example.) In the third step, we detect all such conflicting edges. In the fourth step, we retriangulate all faces $\hat{f}$ so that conflicts are resolved and a final triangulation is obtained.

In [12] we show that, for each face f of G, the face $\hat{f}$, computed in Step 1 of Algorithm 1 is simple. The parts of f that are not in $\hat{f}$ are triangulated. Moreover, Step 1 does not introduce parallel edges. Step 2 triangulates all simple faces $\hat{f}$. However, we may add the same chord $\{v, w\}$ to several faces $\hat{f}_1, \ldots, \hat{f}_k$. It can also happen that $\{v, w\}$ is already an edge of G. If $\{v, w\} \in G$, we have to remove the chords $\{v, w\}$ from all k faces where we have added such a chord. Otherwise, we have to remove $k - 1$ of them. In Step 3, we mark the respective chords as conflicting. We have to show that the output of Step 4 is a conflict-free triangulation of G.

Lemma 5. *Step 4 makes all faces $\hat{f}$ conflict-free, i.e. the graph obtained after Step 4 is simple.*

Proof sketch. Let $d = \{v_0, v_i\}$ (see Fig. 4). Then d cuts $\hat{f}$ into two halves, f_1 and f_2. All chords $\{v_0, v_j\}$, $j < i$ are in f_1; all chords $\{v_0, v_j\}$, $j > i$ are in f_2. That is, f_1 does not contain conflicting chords. Vertex v_{i-1} is the third vertex of the triangle in f_1 that has d on its boundary. Step 4 removes d and all chords in f_2 and retriangulates f_2 with chords incident to v_{i-1}.

Let d' be the edge that is in conflict with d. Then d and d' form a closed curve, and v_{i-1} is outside this curve. All boundary vertices of f_2 excluding the endpoints of d are inside this curve. As no edge, except for the new chords in $\hat{f}$, can intersect this curve, the new chords in $\hat{f}$ are non-conflicting. The "old" chords in $\hat{f}$ were in f_1 and thus, by the choice of d and f_1, non-conflicting. Hence, $\hat{f}$ does not contain conflicting chords. □

We mark the first appearances of vertices in each list F_f in $O(sort(N))$ I/Os as follows: sort F_f by vertex numbers, scan F_f to mark the first appearance of every vertex, and restore the original order of F_f. The rest of Algorithm 1 takes $O(sort(N))$ I/Os.

Lemma 6. *Given the list F as defined in the previous section, Algorithm 1 triangulates the graph G in $O(sort(N))$ I/Os.*

In order to use this algorithm as part of our separator algorithm, we also have to embed the chords in the faces. Let $v_1, e_1, v_2, e_2, \ldots, v_k, e_k$ be the list of vertices and edges visited in a clockwise traversal of the boundary of a face f (i.e., $F_f = \langle v_1, \ldots, v_k \rangle$). We define labels $n_1(v_i) = n_{v_i}(e_{(i-1) \bmod k})$ and $n_2(v_i) = n_{v_i}(e_i)$, and store them with v_i in F_f. When we add a chord d incident to vertex v_i, we give it a label $n_{v_i}(d)$ which is a rational value between $n_1(v_i)$ and $n_2(v_i)$. (To avoid problems related to arithmetic precision, we assign the new label as an offset of $\frac{1}{N}$ from either $n_1(v_i)$ or $n_2(v_i)$.) This embeds d between $e_{(i-1) \bmod k}$ and e_i. After that, the labels $n_1(v_i)$ and $n_2(v_i)$ are updated to ensure that subsequent chords are embedded between $e_{(i-1) \bmod k}$ or e_i and d, depending on the current configuration. We can maintain labels $n_1(v)$ and $n_2(v)$ for all vertices in the lists F_f without increasing the number of I/O-operations by more than a constant factor.

Acknowledgements. We would like to thank Lyudmil Aleksandrov, Jörg-Rüdiger Sack, Hans-Dietrich Hecker, and Jana Dietel for helpful discussions.

References

[1] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, J. S. Vitter. External-memory graph algorithms. *Proc. 6th SODA*, Jan. 1995.

[2] A. Crauser, K. Mehlhorn, U. Meyer. Kürzeste-Wege-Berechnung bei sehr großen Datenmengen. *Aachener Beitr. zur Inf.* (21). Verl. d. Augustinus Buchh. 1997.

[3] F. Dehne, W. Dittrich, D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. *Proc. 9th SPAA*, pp. 106–115, 1997.

[4] H. N. Djidjev. Efficient algorithms for shortest path queries in planar digraphs. *Proc. of the 22nd Workshop on Graph-Theoretic Concepts in Comp. Sci.*, Lecture Notes in Comp. Sci., pp. 151–165. Springer Verlag, 1996.

[5] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comp.*, 16(6):1004–1022, Dec. 1987.

[6] R. J. Lipton, R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36(2):177–189, 1979.

[7] K. Munagala, A. Ranade. I/O-complexity of graph algorithms. *Proc. 10th SODA*, Jan. 1999.

[8] M. Nodine, M. Goodrich, J. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, Aug. 1996.

[9] J. Vitter. External memory algorithms. *Proc. 17th ACM Symp. on Principles of Database Systems*, June 1998.

[10] J. Vitter, E. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

[11] J. Vitter, E. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3):148–169, 1994.

[12] N. Zeh. *An External-Memory Data Structure for Shortest Path Queries.* Diplomarbeit, Fak. f. Math. und Inf., Friedrich-Schiller-Univ. Jena, Nov. 1998.