External Memory Algorithms for Outerplanar Graphs

Anil Maheshwari* and Norbert Zeh

School of Computer Science, Carleton University, Ottawa, Canada
{maheshwa,nzeh}@scs.carleton.ca

Abstract. We present external memory algorithms for outerplanarity testing, embedding outerplanar graphs, breadth-first search (BFS) and depth-first search (DFS) in outerplanar graphs, and finding a $\frac{2}{3}$-separator of size 2 for a given outerplanar graph. Our algorithms take $O(sort(N))$ I/Os and can easily be improved to take $O(perm(N))$ I/Os, as all these problems have linear time solutions in internal memory. For BFS, DFS, and outerplanar embedding we show matching lower bounds.

1 Introduction

Motivation. Outerplanar graphs are a well-studied class of graphs (e.g., see [3, 4, 6]). This class is restricted enough to admit more efficient algorithms than those for general graphs and general enough to have practical applications, e.g. [3].

Our motivation to study outerplanar graphs in external memory is threefold. Firstly, efficient algorithms for triangulating and separating planar graphs were presented in [5, 12]. The major drawback of their separator algorithm is that it requires an embedding and a BFS-tree of the given graph as part of the input. Embedding planar graphs and BFS in general graphs are hard problems in external memory.[1] Our goal is to show that these problems are considerably easier for outerplanar graphs. Secondly, outerplanar graphs can be seen as combinatorial representations of triangulated simple polygons and their subgraphs. Thirdly, every outerplanar graph is a planar graph. Thus, any lower bound that we can show for outerplanar graphs also holds for planar graphs.

Model of computation. When the data set to be handled becomes too large to fit into the main memory of the computer, the transfer of data between fast internal memory and slow external memory (disks) becomes a significant bottleneck. Existing internal memory algorithms usually access their data in a random fashion, thereby causing significantly more I/O operations than necessary. Our goal in this paper is to minimize the number of I/O operations performed. Several computational models for estimating the I/O-efficiency of algorithms have been developed. We adopt the *parallel disk model* PDM [11] as our model of

* Research supported by NSERC and NCE GEOIDE.

[1] No external memory algorithm for embedding planar graphs is known. For BFS in general graphs, the best known algorithm takes $O(|V| + |E|/|V|sort(|V|))$ I/Os [9].

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.

Previous work. For planar graphs, a linear-time algorithm for finding a $\frac{2}{3}$-separator of size $O\left(\sqrt{N}\right)$ was presented in [7]. It is well known that every outerplanar graph has a $\frac{2}{3}$-separator of size 2 and that such a separator can be computed in linear time. Outerplanarity testing [8] and embedding outerplanar graphs take linear time. There are simple linear time algorithms for BFS and DFS in general graphs (see [2]). Refer to [4, 6] for a good exposition of outerplanar graphs.

In the PDM, sorting, permuting, and scanning an array of size N take $sort(N) = \Theta\left(\frac{N}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right)$, $perm(N) = \Theta\left(\min\{N, sort(N)\}\right)$, and $scan(N) = \Theta\left(\frac{N}{DB}\right)$ I/Os [10, 11]. For a comprehensive survey of external memory algorithms, refer to [10]. The best known BFS-algorithm for general graphs takes $O\left(\frac{|E|}{|V|}sort(|V|) + |V|\right)$ I/Os [9]. In [1], $O(sort(N))$ algorithms for computing an open ear decomposition and the connected and biconnected components of a given graph $G = (V, E)$ with $|E| = O(|V|)$ were presented. They also develop a technique, called *time-forward processing*, that can be used to evaluate a directed acyclic graph of size N, viewed as a (logical) circuit, in $O(sort(N))$ I/Os. They apply this technique to develop an $O(sort(N))$ algorithm for list-ranking. An external memory separator algorithm for planar graphs has been presented in [5, 12]; it takes $O(sort(N))$ I/Os provided that a BFS-tree and an embedding of the graph is given. We do not know of any other results for computing separators in external memory efficiently. Also, no efficient external memory algorithms for embedding planar or outerplanar graphs in the plane are known.[2]

Our results. In this paper, we show the following theorem.

Theorem 1. *It takes $O(perm(N))$ I/Os to decide whether a given graph G with N vertices is outerplanar. If G is outerplanar, breadth-first search, depth-first search, and computing an outerplanar embedding of G take $\Theta(perm(N))$ I/Os. Computing a $\frac{2}{3}$-separator of size 2 for G takes $O(perm(N))$ I/Os.*

Preliminaries. A *graph* $G = (V, E)$ is a pair of sets, V and E. V is the *vertex set* of G. E is the *edge set* of G and consists of unordered pairs $\{v, w\}$, where $v, w \in V$. In this paper, $N = |V|$. A *path* in G is a sequence, $P = \langle v_0, \dots, v_k\rangle$, of

[2] One can use the PRAM simulation technique of [1] together with known PRAM results. 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.

vertices such that $\{v_{i-1}, v_i\} \in E$, for $1 \le i \le k$. A graph is *connected* if there is a path between any pair of vertices in G. A graph is *biconnected* if for every vertex $v \in V$, $G - v$ is connected. A *tree* with N vertices is a connected graph with $N - 1$ edges. A *subgraph* of G is a graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. The *connected components* of G are the maximal connected subgraphs of G. The *biconnected components* (bicomps) of G are the maximal biconnected subgraphs of G. A graph is *planar* if it can be drawn in the plane so that no two edges intersect except at their endpoints. This defines an order of the edges incident to every vertex v of G counterclockwise around v. We call G *embedded* if we are given these orders for all vertices of G. $\mathbb{R}^2 \setminus (V \cup E)$ is a set of connected regions. Call these regions the *faces* of G. Denote the set of faces of G by F. A graph is *outerplanar* if it can be drawn in the plane so that there is a face that has all vertices of G on its boundary. For every outerplanar graph, $|E| \le 2|V| - 3$. The *dual* of an embedded planar graph $G = (V, E)$ with face set F is a graph $G^* = (V^*, E^*)$, where $V^* = F$ and E^* contains an edge between two vertices in V^* if the two corresponding faces in G share an edge.

An *ear decomposition* of a biconnected graph G is a decomposition of G into edge-disjoint paths $P_0, \dots, P_k$, $P_i = \langle v_i, \dots, w_i \rangle$, such that $G = \bigcup_{j=0}^{k} P_j$, $P_0 = \langle v_0, w_0 \rangle$, and for every P_i, $i \ge 1$, $P_i \cap G_{i-1} = \{v_i, w_i\}$, where $G_{i-1} = \bigcup_{j=0}^{i-1} P_j$. The paths P_j are called *ears.* An *open ear decomposition* is an ear decomposition such that for every ear P_i, $v_i \ne w_i$.

Let $w : V \to \mathbb{R}^+$ be an assignment of weights to the vertices of G such that $\sum_{v \in V} w(v) \le 1$. The weight of a subgraph of G is the sum of the vertex weights in the subgraph. A $\frac{2}{3}$*-separator* of G is a set S such that none of the connected components of $G - S$ has weight exceeding $\frac{2}{3}$.

We will use the following characterization of outerplanar graphs [4].

Theorem 2. *A graph G is outerplanar if and only if it does not contain a subgraph that is an edge expansion of $K_{2,3}$ or K_4.*

In our algorithms we represent a graph G as the two sets V and E. The embedding of a graph is represented as labels $n_v(e)$ and $n_w(e)$ for every edge $e = \{v, w\}$, where $n_v(e)$ and $n_w(e)$ are the positions of e in the counterclockwise orders of the edges around v and w, respectively.

2 Embedding and Outerplanarity Testing

We show how to compute a combinatorial embedding (i.e., edge labels $n_v(e)$ and $n_w(e)$) of a given outerplanar graph G. Our algorithm for outerplanarity testing is based on the embedding algorithm. We restrict ourselves to biconnected outerplanar graphs. If the given graph is not biconnected, we compute the biconnected components in $O(\mathit{sort}(N))$ I/Os [1]. Then we compute embeddings of the biconnected components and join them at the cutpoints of the graph.

Our algorithm for embedding biconnected outerplanar graphs G consists of two steps. In Step 1 we compute the cycle C in G that represents G's outer

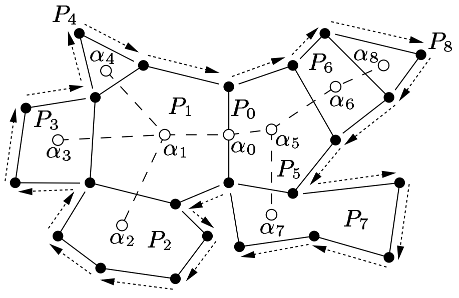

Fig. 1

boundary in the embedding. In Step 2 we embed the remaining edges, which are diagonals of C. Step 1 relies on Observation 1. Let $\mathcal{E}_G = \langle P_0, \ldots, P_k \rangle$ be the given open ear decomposition. We call an ear P_i, $i \geq 1$, *trivial* if it consists of a single edge. Otherwise we call it *non-trivial*. Ear P_0 is non-trivial by definition. Let $G_i = \bigcup_{j=0}^{i} P_j$. Given an embedding of G_i, we call two vertices, v and w, of G_i *consecutive* if there is an edge $\{v, w\}$ in G_i that is on the outer boundary of G_i.

Observation 1. *Given a decomposition of a biconnected outerplanar graph G into open ears $P_0, \ldots, P_k$ and an embedding of G. Then either P_i is trivial or the endpoints of P_i are consecutive in G_{i-1}, for $1 \leq i \leq k$.*

Observation 1 implies that, except if a non-trivial ear P_i is attached to the endpoints of P_0, there is exactly one non-trivial ear P_j, $j < i$, that contains both endpoints of P_i. We represent this relationship between non-trivial ears in the *ear tree* T_E of G (see Fig. 1). This tree contains a node α_k for every non-trivial ear P_k. Node α_j is the parent of node α_i ($\alpha_j = p(\alpha_i)$) if P_j contains both endpoints of P_i. If P_i's endpoints are also P_0's endpoints, α_i is the child of α_0. The vertices in P_i must appear in the same order along the outer boundary of G as they appear in P_i. Using these observations, we construct the order of the vertices along C using a depth-first traversal of T_E as follows:

Start the traversal of T_E at node α_0. At a node α_i, we traverse the ear P_i and append P_i's vertices to C. When we reach an endpoint of an ear P_j with $\alpha_i = p(\alpha_j)$, we recursively traverse the subtree rooted at α_j. When we are done with the traversal of that subtree, we continue traversing P_i.

To construct the final embedding, we use the following observation: Let $e_1, \ldots, e_d$ be the edges incident to a vertex v, sorted clockwise around v. Let u_i be the other endpoint of edge e_i, for $1 \leq i \leq d$. Then the vertices $u_1, \ldots, u_d$ appear in clockwise order along the outer boundary of G (see Fig. 1).

Lemma 1. *An outerplanar embedding of a given outerplanar graph G with N vertices can be computed in $O(sort(N))$ I/Os.*

Proof sketch. Open ear decomposition: The open ear-decomposition of G can be computed in $O(sort(N))$ I/Os [1]. We scan the list, $\mathcal{E}_G$, of ears to construct the list, $\mathcal{E}'_G$, of non-trivial ears.
Ear tree construction: Let v be a vertex that is interior to ear P_i. (Note that every vertex, except for the two endpoints of P_0, is interior to exactly one ear.) Then we define v's ear number as $\epsilon(v) = i$. For the two endpoints, v_0 and w_0, of P_0 we define $\epsilon(v_0) = \epsilon(w_0) = 0$. Let $\alpha_i = p(\alpha_j)$ in T_E. It can be shown that $i = \max\{\epsilon(v_j), \epsilon(w_j)\}$. We scan the list of non-trivial ears to compute all ear numbers. The ear tree can then be constructed in $O(sort(N))$ I/Os.
Construction of C: Consider the ears corresponding to the nodes stored in a subtree $T_E(\alpha_j)$ of T_E rooted at a node α_j. The internal vertices of these ears are exactly the vertices that appear between the two endpoints, v_j and w_j, of P_j on the outer boundary of G, i.e., in C. The number of these vertices can be computed as follows: We assign the number of internal vertices of ear P_j as a weight $w(\alpha_j)$ to every node α_j and use time-forward processing to compute the subtree weights $w(T_E(\alpha_j))$ of all subtrees $T_E(\alpha_j)$.

Now we sort the ears P_i in $\mathcal{E}'_G$ by their indices i. Scanning this sorted list of ears corresponds to processing T_E from the root to the leaves. We use time-forward processing to send small pieces of additional information down the tree. We start at node α_0 and assign $n(v_0) = 0$ and $n(w_0) = w(T_E(\alpha_1)) + 1$, where $n(v)$ is v's position in a clockwise traversal of C. For every subsequent ear, P_j, we maintain the invariant that when we start the scan of P_j, we have already computed $n(v_j)$ and $n(w_j)$. Then we scan along P_j and number the vertices of P_j in their order of appearance. Let x be the current vertex in this scan and y be the previous vertex. If there is no ear attached to x and y, $n(x) = n(y) + 1$. Otherwise, let ear P_i be attached to x and y (i.e., $\{x, y\} = \{v_i, w_i\}$). Then $n(x) = n(y) + w(T_E(\alpha_i)) + 1$. We send $n(v_i)$ and $n(w_i)$ to α_i so that this information is available when we process P_i. Note that the current ear P_j might be stored in reverse order (i.e., $n(w_j) < n(v_j)$). This can easily be handled using a stack to reverse P_j before scanning.
Computing the final embedding: First, we relabel the vertices of G in their order around C. Then we replace every edge $e = \{v, w\}$ by two directed edges (v, w) and (w, v). We sort the list of these edges lexicographically and scan it to compute the desired labels $n_v(e)$ and $n_w(e)$. □

Note that ear P_0 requires some special treatment because it is the only ear that can have two non-trivial ears attached on two different sides. However, the details are fairly straightforward and therefore omitted. The above algorithm can be augmented to do outerplanarity testing in $O(sort(N))$ I/Os. Due to space constraints, we refer the reader to the full version of the paper.

3 Breadth-first and Depth-first Search

We can restrict ourselves to BFS and DFS in biconnected outerplanar graphs. If the graph is not biconnected, we compute its connected and biconnected components in $O(sort(N))$ I/Os [1]. Then we represent every connected component

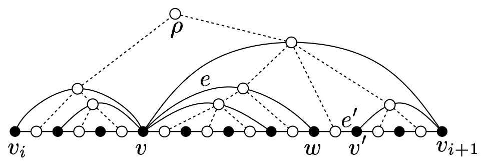

Fig. 2

by a rooted tree describing the relationship between its bicomps and cutpoints. We process each such tree from the root to the leaves, applying Lemmas 3 and 2 to the bicomps of the graph, in order to compute a BFS resp. DFS-tree for every connected component.

Given a biconnected outerplanar graph G, we first embed it. The list C computed by our embedding algorithm contains the vertices of G sorted counterclockwise along the outer boundary of G. Given a source vertex r, a path along the outer boundary of G, starting at r, is a DFS-tree of G. Thus, we can compute a DFS-tree of G by scanning C.

Lemma 2. *Given a biconnected outerplanar graph G with N vertices, depth-first search in G takes $O(sort(N))$ I/Os.*

The construction of a BFS-tree is based on the following observation: Let the vertices of G be numbered counterclockwise around the outer boundary of G and such that the source, r, has number 1. Let $v_1 < \cdots < v_k$ be the neighbours of r. The removal of these vertices partitions G into subgraphs G_i, $1 \leq i < k$, induced by vertices $v_i + 1, \ldots, v_{i+1} - 1$. Indeed, if there was an edge $\{v, w\}$ between two such graphs G_i and G_j, $i < j$, G would contain a K_4 consisting of the paths between r, v, w, v_i, v_j, and v_{j+1}. We consider graphs $\tilde{G}_i$ that are induced by vertices $v_i, \ldots, v_{i+1}$. Let e and e' be two edges in such a graph $\tilde{G}_i$ such that the left endpoint of e is to the left of the left endpoint of e'. Then either e is completely to the left of e' or e spans e'.

The vertices v_i and v_{i+1} are at distance 1 from the root r, and the shortest path from any vertex in $\tilde{G}_i$ to r must contain v_i or v_{i+1}. Thus, we can do BFS in $\tilde{G}_i$ by finding the shortest path from every vertex in $\tilde{G}_i$ to either v_i or v_{i+1}, whichever is shorter. We build an *edge tree* T_e for $\tilde{G}_i$ (see Fig. 2). T_e contains a node $v(e)$ for every edge e in $\tilde{G}_i$. Node $v(e)$ is the parent of another node $v(e')$ if edge e spans edge e' and there is no edge e'' that spans e' and is spanned by e. T_e has an additional root node ρ, which is the parent of all nodes $v(e)$, where edge e is not spanned by any other edge. The *level* of an edge e in $\tilde{G}_i$ is the distance of node $v(e)$ from the root ρ of T_e.

We call an edge $e = \{v, w\}$ incident to vertex w a *left* (resp. *right*) edge if $v < w$ (resp. $v > w$). The minimal left (resp. right) edge for w has the minimal level in T_e among all left (resp. right) edges incident to w. The shortest path from w to v_i (v_{i+1}) must contain either the minimal left edge, $e = \{v, w\}$, or the minimal right edge, $e' = \{v', w\}$, incident to w (see Fig. 2). Indeed, the shortest path must contain v or v', and e (resp. e') is the shortest path from

w to v (resp. v'). Thus, we can process T_e level by level from the root towards the leaves maintaining the following invariant: When we visit a node $v(e)$ in T_e, where $e = \{v, w\}$, we know the distances $d(r, v)$ and $d(r, w)$ from v and w to the source vertex r. Assume that $v(e)$'s children are sorted from left to right. We scan the list of the corresponding edges from left to right to compute the distances of their endpoints to v. In a right-to-left scan we compute the distances to w. Then it takes another scan to determine for every edge endpoint x, the distance $d(r, x) = \min\{d(r, v) + d(v, x), d(r, w) + d(w, x)\}$ and to set the parent pointers in the BFS-tree properly. This computation ensures the invariant for $v(e)$'s children.

Lemma 3. *Given a biconnected outerplanar graph G with N vertices, breadth-first search in G takes $O(sort(N))$ I/Os.*

Proof sketch. Constructing $\tilde{G}_1, \dots, \tilde{G}_{k-1}$: We use our embedding algorithm to embed G and compute the order of the vertices of G along the outer boundary of G. Given this ordering, we use sorting and scanning to split G into subgraphs $\tilde{G}_i$.
Constructing T_e: We sort the list, V_i, of vertices of $\tilde{G}_i$ from left to right and store with every vertex w, the list, $E(w)$, of edges incident to w, sorted clockwise around w. Then we scan V_i and apply a simple stack algorithm to construct T_e: We initialize the stack by pushing ρ on the stack. When visiting vertex w, we first pop the nodes $v(e)$ for all left edges $e = \{v, w\}$ in $E(w)$ from the stack and make the next node on the stack the parent of the currently popped node. Then we push the nodes $v(e)$ for all right edges in $E(w)$ on the stack, in their order of appearance.
Breadth-first search in $\tilde{G}_i$: We sort the edges of $\tilde{G}_i$ by increasing level and from left to right and use time-forward processing to send the computed distances downward in T_e. Scanning the list of children of the currently visited node $v(e)$ takes $O(scan(N))$ I/Os for all nodes of T_e because the edges of every level are sorted from left to right, and the right-to-left scans can be implemented using a stack. □

4 Separating Outerplanar Graphs

We assume that G is connected and no vertex in G has weight greater than $\frac{1}{3}$. If there is a vertex v with weight $w(v) > \frac{1}{3}$, $S = \{v\}$ is trivially a $\frac{2}{3}$-separator of G. If G is disconnected, we compute G's connected components in $O(sort(N))$ I/Os [1]. If there is no component of weight greater than $\frac{2}{3}$, $S = \emptyset$ is a $\frac{2}{3}$-separator of G. Otherwise, we compute a separator of the connected component of weight greater than $\frac{2}{3}$.

Our strategy for finding a size-2 separator of G is as follows: First we embed G and make G biconnected by adding appropriate edges to the outer face of G. Then we triangulate the interior faces of the resulting graph and compute the dual tree T^* corresponding to the interior faces of the triangulation $G_\triangle$ of G. Every edge in T^* corresponds to a diagonal of the triangulation, whose two

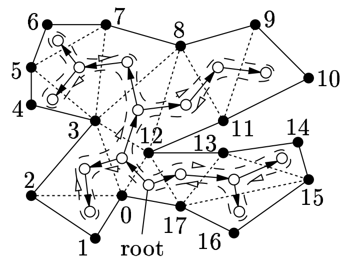

Fig. 3

endpoints are a size-2 separator of $G_\triangle$ and thus of G. We assign appropriate weights to the vertices of T^* that allow us to find a tree edge that corresponds to a $\frac{2}{3}$-separator of $G_\triangle$.

We use the following observations: (1) G is biconnected if and only if its outer face is simple. (A face is simple, if every vertex appears at most once in a counterclockwise traversal of its boundary.) (2) If G is a cycle, we can choose one of the two faces of G as the outer face. Otherwise, the outer face of G is the only face that has all vertices of G on its boundary.

The triangulation algorithm for planar graphs in [5] first makes all faces of the given graph simple and then triangulates the resulting simple faces. Using the two observations just made, this triangulation algorithm can easily be modified to triangulate all faces of G except the outer face, which is only made simple.

Lemma 4. *A size-2 $\frac{2}{3}$-separator of a given outerplanar graph G with N vertices can be computed in $O(sort(N))$ I/Os.*

Proof sketch. We have to show how to construct the dual tree T^* corresponding to the interior faces of $G_\triangle$ and how to find an edge of T^* corresponding to a $\frac{2}{3}$-separator of $G_\triangle$.
Constructing the dual tree: We first number the vertices 0 through $N-1$ clockwise around $G_\triangle$. With every vertex v we store its adjacency list $A(v)$ sorted counterclockwise around v, where $(v-1) \bmod N$ is the first vertex in $A(v)$. Denote the concatenation of $A(0), \ldots, A(N-1)$ by A. We construct T^* recursively (see Fig. 3). We start at edge $\{a, b\} = \{0, N-1\}$ and consider triangle $\triangle_1 = (a, b, x)$.

We make the vertex $v(\triangle_1)$ corresponding to $\triangle_1$ the root of T^*. Let $\triangle_2 = (a, x, y)$ and $\triangle_3 = (x, b, z)$ be the two triangles adjacent to $\triangle_1$. Then the corresponding vertices $v(\triangle_2)$ and $v(\triangle_3)$ are the children of $v(\triangle_1)$. We recursively construct the subtrees of T^* rooted at $v(\triangle_2)$ and $v(\triangle_3)$, calling the tree construction procedure with parameters $\{a_1, b_1\} = \{a, x\}$ and $\{a_2, b_2\} = \{x, b\}$, respectively. Using this strategy we basically perform a depth-first traversal of T^*. The corresponding Euler tour (as represented by the dashed line in Fig. 3) crosses the edges of $G_\triangle$ in their order of appearance in A. Thus, T^* can be constructed in a single scan over A and using $O(N)$ stack operations.

Finding the separator: At every recursive call we assign weights $w(v(\triangle_1)) = w(x)$ and $w_p(v(\triangle_1)) = w(a) + w(b)$ to the newly created vertex $v(\triangle_1)$. Let $T^*(v)$ be the subtree of T^* rooted at a vertex v. During the construction of T^* we can compute for every vertex v, the weight $w(T^*(v)) = \sum_{u \in T^*(v)} w(u)$ of the subtree $T^*(v)$. The removal of the edge connecting $v(\triangle_1)$ to its parent in T^* corresponds to removing vertices a and b from $G_\triangle$. This partitions $G_\triangle$ into two subgraphs of weights $w(T^*(v(\triangle_1)))$ and $w(G_\triangle) - w(T^*(v(\triangle_1))) - w_p(v(\triangle_1))$. Thus, once the weights $w(T^*(v))$ and $w_p(v)$ have been computed for every vertex v of T^*, it takes a single scan over the vertex list of T^* to compute a size-2 $\frac{2}{3}$-separator of $G_\triangle$ and thus of G. □

5 Lower Bounds

In this section, we prove matching lower bounds for all results in this paper, except for computing separators. We show these lower bounds by reducing list-ranking, which has an $\Omega(perm(N))$ lower bound [1], to DFS, BFS, and embedding of biconnected outerplanar graphs. The list-ranking problem is defined as follows: *Given a singly linked list L and a pointer to the head of L, compute for every node of L its distance to the tail.*
Note that list-ranking reduces trivially to DFS and BFS in general outerplanar graphs, as we can consider the list itself as an outerplanar graph and choose the tail of the list as the source of the search.

Lemma 5. *List-ranking can be reduced to computing a combinatorial embedding of a biconnected outerplanar graph in $O(scan(N))$ I/Os.*

Proof sketch. Given the list L, we can compute the tail t of L (t is the node with no successor) and node t' with $succ(t') = t$ in two scans over L. We consider L as a graph with an edge between every vertex and its successor. In another scan, we add edges $\{v, t\}$ to L, for $v \notin \{t, t'\}$. This gives us a graph G_1 as in Fig. 4(a). It can be shown that the outerplanar embedding of G_1 is unique except for flipping the whole graph. Thus, the rank of v in L is the position of edge $\{v, t\}$ in clockwise or counterclockwise order around t. □

Lemma 6. *List-ranking can be reduced to breadth-first search and depth-first search, respectively, in biconnected outerplanar graphs in $O(scan(N))$ I/Os.*

Proof sketch. We only show the reduction for BFS. The reduction to DFS is even simpler. As in the proof of the previous lemma, we identify the tail t of L in a single scan. Then we add an edge $\{h, t\}$ to L. This produces a cycle G_2. We perform two breadth-first searches (see Fig. 4(b)), one with source t (labels outside) and one with source h (labels inside). It is easy to see that the distance $d(v)$ of every node to the tail of the list can now be computed as $d(v) = N - 1 - d(h, v)$ if $d(h, v) < d(t, v)$, and $d(v) = d(t, v)$ otherwise. □

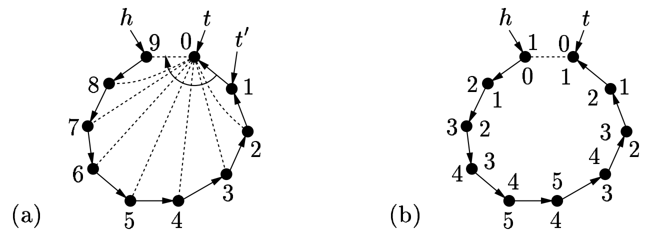

Fig. 4

Proof sketch (Theorem 1). The theorem follows from the lemmas in this paper, if we can reduce the upper bounds from $O(sort(N))$ to $O(perm(N))$ I/Os. Let $\mathcal{P}$ be the problem at hand, $\mathcal{A}$ be an $O(N)$ time internal memory algorithm that solves $\mathcal{P}$, and $\mathcal{A}'$ be an $O(sort(N))$ external memory algorithm that solves $\mathcal{P}$. (For the problems studied in this paper, linear time solutions in internal memory are known, and we have provided the $O(sort(N))$ algorithms.) We run algorithms $\mathcal{A}$ and $\mathcal{A}'$ in parallel, switching from one algorithm to the other at every I/O operation. The computation stops as soon as one of the algorithms terminates. At this point algorithms $\mathcal{A}$ and $\mathcal{A}'$ have performed at most $\min\{O(N), O(sort(N))\} = O(perm(N))$ I/Os. □

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. T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.
3. G. N. Frederickson. Searching among intervals in compact routing tables. *Algorithmica*, 15:448–466, 1996.
4. F. Harary. *Graph Theory.* Addison-Wesley, 1969.
5. D. Hutchinson, A. Maheshwari, N. Zeh. An external memory data structure for shortest path queries. *Proc. COCOON'99*, LNCS 1627, pp. 51–60, July 1999.
6. J. van Leeuwen. *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity.* MIT Press, 1990.
7. R. J. Lipton, R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. on Applied Mathematics*, 36(2):177–189, 1979.
8. S. L. Mitchell. Linear algorithms to recognize outerplanar and maximal outerplanar graphs. *Inf. Proc. Letters*, 9(5):229–232, Dec. 1979.
9. K. Munagala, A. Ranade. I/O-complexity of graph algorithms. *Proc. 10th SODA*, Jan. 1999.
10. J. S. Vitter. External memory algorithms. *Proc. 17th ACM Symp. on Principles of Database Systems*, June 1998.
11. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 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.