On External-Memory MST, SSSP and Multi-way Planar Graph Separation

(Extended Abstract)

Lars Arge[1,*], Gerth Stølting Brodal[2,**], and Laura Toma[1,***]

[1] Duke University, Durham, NC 27708–0129, USA
[2] University of Aarhus, DK-8000 Århus C, Denmark

Abstract. Recently external memory graph algorithms have received considerable attention because massive graphs arise naturally in many applications involving massive data sets. Even though a large number of I/O-efficient graph algorithms have been developed, a number of fundamental problems still remain open. In this paper we develop an improved algorithm for the problem of computing a minimum spanning tree of a general graph, as well as new algorithms for the single source shortest paths and the multi-way graph separation problems on planar graphs.

1 Introduction

Recently external memory graph algorithms have received considerable attention because massive graphs arise naturally in many applications involving massive data sets. One example of a massive graph is AT&T's 20TB phone-call data graph [11]. Other examples of massive graphs arise in Geographic Information Systems (GIS). For instance, GIS terrains are often represented using planar graphs and many common GIS problems can be formulated as standard graph problems (Arc/Info [4], the most commonly used GIS package, contains functions that correspond to computing depth-first, breadth-first, and minimum spanning trees, as well as shortest paths and connected components). When working with such massive graphs the I/O-communication, and not the internal memory computation time, is often the bottleneck. Designing efficient external memory algorithms for such problems can thus lead to considerable runtime improvements, as for example illustrated in our previous work [7].

Even though a large number of I/O-efficient graph algorithms have been developed in recent years, a number of important problems still remain open. For example, developing efficient algorithms for basic problems such as breadth-first

* Supported in part by the National Science Foundation through ESS grant EIA–9870734, RI grant EIA-9972879 and CAREER grant EIA-9984099. E-mail: large@cs.duke.edu.

** BRICS (Basic Research in Computer Science, Center of Danish National Research Foundation). Supported in part by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT). E-mail: gerth@brics.dk.

*** Supported in part by the National Science Foundation through ESS grant EIA–9870734 and RI grant EIA-9972879. E-mail: laura@cs.duke.edu.

search and depth-first search remain open. In this paper we develop I/O-efficient algorithms for the minimum spanning tree (MST) and single source shortest paths (SSSP) problems, as well as for multi-way planar graph separation.

1.1 Problem Statement

MST and SSSP are well-known problems on a weighted graph $G = (V, E)$: MST is the problem of finding a spanning tree for G of minimum weight and SSSP is the problem of finding the shortest paths from a given source vertex in G to all other vertices in G (the length of a path is the sum of the weights of the edges on the path).

Consider an undirected graph $G = (V, E)$.[1] An *$f(V)$-separator* of G is a subset S of the vertices of G of size $f(V)$ such that the removal of S disconnects G into two subgraphs G_1 and G_2, each of size at most $\frac{2V}{3}$. Lipton and Tarjan [23] proved that any planar graph has an $O(\sqrt{V})$-separator and gave a linear time algorithm for finding such a separator. Using this result recursively, a planar graph can be decomposed into $\Theta(\frac{V}{R})$ subgraphs G_i with $O(R)$ vertices each and $O(\frac{V}{\sqrt{R}})$ separator vertices, such that there is no edge between a vertex in G_i and a vertex in G_j for $i \neq j$. We call such a decomposition a *multi-way planar graph separation* of G. Graph separation is often used in the design of divide-and-conquer algorithms.

Throughout this paper we assume that the input graph G is given in edge-list representation. If G is planar we assume it is embedded in the plane. We also assume without loss of generality that G is connected and that no two edges have the same weight. In some of our algorithms we will assume that a breadth-first-search tree T of G is given. In such cases we assume that T is represented implicitly by storing with each vertex u in G its parent in T and marking every edge of G as either a tree or a non-tree edge.

1.2 Previous Results on I/O-efficient Graph Algorithms

We work in the standard two-level I/O model with one (logical) disk [3, 20]. The model defines the following parameters:

$N = V + E,$

$M =$ number of vertices/edges that can fit into internal memory,

$B =$ number of vertices/edges per disk block,

where $M < N$ and $1 \leq B \leq M^{1/(2+\varepsilon)}$, for some $\varepsilon > 0$.[2] An *Input/Output* (or simply *I/O*) involves reading (or writing) a block from disk into (from) internal memory. Our measure of performance of an algorithm is the number of I/Os it performs. The number of I/Os needed to read N contiguous items from disk is $\text{scan}(N) = \Theta(\frac{N}{B})$ (the scanning bound), and the number of I/Os required to

[1] For convenience we will use the name of a set to denote both the actual set and its cardinality.

[2] Often it is only assumed that $B \leq M/2$ but sometimes, as in this paper, the very realistic assumption that the main memory is capable of holding B^2 elements is made (or as here, $B^{2+\varepsilon}$ for some $\varepsilon > 0$).

sort N items is $\text{sort}(N) = \Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ [3] (the sorting bound). In practice the difference between an algorithm doing N I/Os and one doing $\text{scan}(N)$ or $\text{sort}(N)$ I/Os can be significant [7].

Table 1. Best known upper bounds for basic graph theoretic problems.

Problem	General undirected graphs	
DFS	$O\left(\frac{V}{M}\frac{E}{B} + V\right)$	[12]
	$O\left((V + \text{scan}(E)) \cdot \log\frac{V}{B} + \text{sort}(E)\right)$	[22]
BFS	$O(V + \frac{E}{V} \cdot \text{sort}(V))$	[25]
CC	$O\left(\text{sort}(E) \cdot \log\log\frac{VB}{E}\right)$	[25]
MST	$O\left(\text{sort}(E) \cdot \log\frac{V}{M}\right)$	[12]
	$O\left(\text{sort}(E) \cdot \log B + \text{scan}(E) \cdot \log V\right)$	[22]
SSSP	$O\left(V + \frac{E}{B} \cdot \log\frac{V}{B}\right)$	[22]

I/O-efficient graph algorithms have been considered by a number of authors [1, 2, 5, 6, 10, 12, 16, 19, 22, 24–26, 29]. Table 1 reviews the best known algorithms for basic graph theoretic problems on general undirected graphs. For directed graphs the best known algorithm for breadth-first search (BFS) and depth-first search (DFS) use $O\left((V + \text{scan}(E)) \cdot \log\frac{V}{B} + \text{sort}(E)\right)$ I/Os [10]. Lower bound results were proved in [6, 12, 25]. Note that no $O(\text{sort}(E))$ (deterministic) algorithm is known for *any* of the problems, and that the best known algorithms for DFS, BFS and SSSP require $\Omega(V)$ I/Os. MST and connected components (CC) can be solved in $O(\text{sort}(E))$ I/Os with randomized algorithms [12, 1].

Improved algorithms have been developed for several special classes of graphs. For trees, $O(\text{sort}(N))$ algorithms are known for BFS and DFS numbering, Euler tour computation, expression tree evaluation, topological sorting, as well as several other problems [10, 12]. For planar graphs, $O(\text{sort}(N))$ algorithms are known for CC and MST [12]. For grid graphs $O(\text{sort}(N))$ algorithms are known for BFS and SSSP, and an $O(\text{scan}(N))$ algorithm for CC [7]. See [30] for a complete reference.

Given that even very basic graph problems seem hard to externalize, it is natural to try to reduce the problems to one another. A first step in this direction was taken by Hutchinson *et al.* [19] who considered the problem of computing an $O(\sqrt{N})$-separator of a planar graph I/O-efficiently. Given a BFS tree they showed how to compute a separator in $O(\text{sort}(N))$ I/Os. Given this algorithm, it is straightforward to solve the multi-way planar graph separation problem in $O(\log\frac{N}{R} \cdot \text{sort}(N)))$ I/Os, simply by applying the algorithm recursively.

1.3 Our results

In Section 2, we give an $O(\text{sort}(E) \cdot \log\log\frac{VB}{E}) = O(\text{sort}(E) \cdot \log\log B)$ algorithm for the MST problem on general undirected weighted graphs, improving the previous bound of $O\left(\text{sort}(E) \cdot \log B + \text{scan}(E) \cdot \log V\right)$ [22]. The algorithm uses the same general idea as the CC algorithm by Munagala and Ranade [25] and consists of two phases: first a vertex contraction algorithm is used to reduce the number of vertices to $O(\frac{E}{B})$, and then an $O(V + \text{sort}(E))$ MST algorithm is used on the reduced graph. The new contraction algorithm uses ideas similar to the ones used in [8, 14, 25], as well as a simplified version of the basic contraction step used in previous MST algorithms [8, 12–14, 22, 25, 28]. The new $O(V + \text{sort}(E))$ MST algorithm is a modified version of Prim's algorithm. It remains a challenging open problem to develop an $O(\text{sort}(E))$ MST algorithm.

In Section 3 and 4, we show that the multi-way planar graph separation problem and the SSSP problem can be reduced to the BFS problem in $O(\text{sort}(N))$ I/Os: In Section 3, we give an $O(\text{sort}(N))$ algorithm for the multi-way planar graph separation problem given a BFS tree. The algorithm improves the straightforward bound of $O(\log \frac{N}{R} \cdot \text{sort}(N))$ I/Os and uses a divide-and-conquer algorithm based on ideas from [18]. In Section 4, we show how to use this result to solve the SSSP problem in $O(\text{sort}(N))$ I/Os. The algorithm is a generalization of our SSSP algorithm on grid graphs [7] and uses ideas similar to the ones utilized by Frederickson [17]. We believe that our $O(\text{sort}(N))$ graph separation algorithm might prove helpful in reducing other problems on planar graphs to the BFS problem. It remains a challenging problem to develop an $O(\text{sort}(E))$ BFS algorithm. Another interesting open problem is if it is possible to develop an $O(\text{sort}(E))$ BFS algorithm for a planar graph given a multi-way separation of the graph.

2 Minimum Spanning Tree on General Graphs

In this section we describe our MST algorithm on general undirected weighted graphs. The basic idea is to reduce the number of vertices to $\frac{E}{B}$ using an $O(\text{sort}(E))$ vertex reduction algorithm $O(\log\log \frac{VB}{E})$ times, and then use an $O(V + \text{sort}(E))$ MST algorithm on the resulting graph. The overall I/O complexity will thus be $O(\text{sort}(E) \cdot \log\log \frac{VB}{E} + \frac{E}{B} + \text{sort}(E)) = O(\text{sort}(E) \cdot \log\log \frac{VB}{E})$ I/Os. In Section 2.1 we first describe the $O(V + \text{sort}(E))$ MST algorithm, and in Section 2.2 we then describe the reduction algorithm. The MST result is summarized in the following theorem.

Theorem 1. *The MST of an undirected weighted graph can be found in* $O(\text{sort}(E) \cdot \log\log \frac{VB}{E})$ *I/Os.*

2.1 An $O(V + \text{sort}(E))$ MST Algorithm

Our algorithm is a modified version of Prim's internal memory algorithm [15]. The idea of Prim's algorithm is to grow the MST iteratively from a source node while maintaining a priority queue on the vertices not included in the MST so far; the priority of a vertex is the weight of the minimum edge connecting it to the current MST. The algorithm repeatedly extracts the minimum priority vertex v, adds it to the MST, and updates the priority of the vertices u adjacent to v. Specifically, the weight w of edge (v, u) is compared with the priority of vertex u in the priority queue, and an update is performed if w is smaller than the current priority. Prim's algorithm cannot be implemented efficiently in external memory, the main reason being that the current priority of a given vertex cannot in general be obtained without doing one I/O. A direct implementation would thus lead to an $O(E)$ I/O bound. Previously known algorithms [12, 22] rely instead on vertex contraction methods [8, 13, 14].

Our modification of Prim's algorithm consists of storing edges in the priority queue instead of vertices. During the algorithm the priority queue contains (at least) all edges connecting vertices in the current MST with vertices not in the tree. The queue can also contain edges between two vertices in the MST. The algorithm works as follows: Repeatedly perform *extract_min* to extract the minimum weight edge (u, v) from the priority queue. If v is already in the MST the edge is discarded. Otherwise v is included in the MST and all edges incident to v, except (v, u), are inserted in the priority queue. The key to the I/O-efficiency of the algorithm is that because we store edges in the priority queue we have a simple way of checking whether a vertex is already included in MST — as all edges incident to v are inserted in the priority queue when v is included in the MST, it follows that if both u and v are in the MST when processing an edge $e = (u, v)$, the edge e must appear in the priority queue twice. Thus we can check if v is already included in the MST simply by performing one more *extract_min* and checking if it returns the same edge e (recall that we assume that no two edges have the same weight).

The algorithm performs at least one I/O for each vertex which is included in the MST in order to read its adjacent vertices (traverse its adjacency lists). Thus processing all vertices and edges takes $V + \frac{E}{B}$ I/Os. It also performs $O(E)$ *insert*'s and *extract_min*'s on the priority queue. Using an external priority queue [5, 9] supporting these operations in $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/Os amortized we obtain:

Lemma 1. *The MST of an undirected weighted graph can be computed in* $O(V + \text{sort}(E))$ *I/Os.*

2.2 MST Vertex-Reduction Algorithm

Our MST vertex reduction algorithm is obtained using ideas from the connected-component algorithm of Munagala and Ranade [25] and the notion of "blocking values". The standard MST algorithm based on vertex contraction proceeds in $\lceil \log V \rceil$ phases [12, 22]. In each phase the minimum cost edge adjacent to every vertex v is selected and output as part of the MST and the vertices connected by the selected edges are contracted to supervertices. Let the size of a supervertex be the number of vertices it contains from the original graph. After the ith phase the size of every supervertex is at least 2^i. Since one contraction phase can be performed in $O(\text{sort}(E))$ I/Os [12] this results in an $O(\text{sort}(E) \cdot \log V)$ algorithm. The algorithm in [22] utilizes that a contraction step can be performed more efficiently after $O(\log B)$ phases and obtains an $O(\text{sort}(E) \cdot \log B + \text{scan}(E) \cdot \log V)$ algorithm.

Our algorithm runs for $\lceil \log \frac{VB}{E} \rceil$ phases after which the number of supervertices is at most $\frac{E}{B}$. Furthermore we reduce the number of I/Os used in the process by dividing the $\lceil \log \frac{VB}{E} \rceil$ phases into *superphases* requiring $O(\text{sort}(E))$ I/Os each: Let $N_i = 2^{(3/2)^i}$, i.e. $N_{i+1} = N_i \sqrt{N_i}$. Superphase i, for $i \geq 0$, consists of $\lceil \log \sqrt{N_i} \rceil$ phases. In a preprocessing step we run the basic vertex contraction algorithm once to insure that the number of vertices before superphase 0 is $V_0 \leq \frac{V}{N_0} = \frac{V}{2}$. We will maintain the invariant that before superphase i the number of supervertices is at most $\frac{V}{N_i}$. To reduce the number of vertices to at most $\frac{E}{B}$ it is therefore sufficient to perform $3 + \lceil \log_{3/2} \lceil \log \frac{VB}{E} \rceil \rceil$ superphases and we obtain the $O(\text{sort}(E) \cdot \log\log \frac{VB}{E})$ algorithm.

The phases in each superphase only work on a subset of the (remaining) edges. The edge subsets are chosen in order to allow each supervertex to grow by a factor of $\sqrt{N_i}$ in superphase i. Let $G_i = (V_i, E_i)$ be the graph just prior to superphase i. We construct a graph $G'_i = (V_i, E'_i)$, where E'_i is a subset of E_i. For each vertex v, E'_i contains the $\lceil \sqrt{N_i} \rceil$ lightest edges adjacent to v. Heavier edges $e = (v, u)$ adjacent to v are only included in E'_i if e is among the $\lceil \sqrt{N_i} \rceil$ lightest edges adjacent to u. We define the *blocking value* of v to be the weight of the $(\lceil \sqrt{N_i} \rceil + 1)$-th lightest edge adjacent to v. The set E'_i and blocking values can be computed using $O(\text{sort}(E_i))$ I/Os. If we guarantee that $V_i \leq \frac{V}{N_i}$ as stated above, it follows that $E'_i \leq 2V_i \lceil \sqrt{N_i} \rceil < 4\frac{V}{\sqrt{N_i}}$. As each contraction phase in superphase i can be performed in $O(\text{sort}(E'_i))$ I/Os, it follows that superphase i requires $O(\text{sort}(E_i) + \text{sort}(E'_i) \cdot \log(\sqrt{N_i})) = O(\text{sort}(E) + \text{sort}(\frac{V}{\sqrt{N_i}}) \cdot \log(\sqrt{N_i})) = O(\text{sort}(E))$ I/Os. After performing all the phases of superphase i the edges $E_i - E'_i$, i.e. the heavy edges which were not included in the sample, need to be re-incorporated in E_{i+1}. This can be easily be done as in [25] using $O(\text{sort}(E))$ I/Os in total. Details will appear in the full paper.

The only thing that remains to be described is how the individual phases in superphase i are performed such that after superphase i the number of supervertices is at most $\frac{V}{N_{i+1}}$ and such that only edges that actually belong to the MST are included. A phase is performed as in the basic vertex reduction algorithm: For each vertex v consider the adjacent edge e with minimum weight in E'_i. If the weight of e is smaller than the blocking value of v, then we select e for contraction. If the weight of e is larger than the blocking value, no edges is selected for v, since there might be a lighter edge adjacent to v in $E_i - E'_i$. The selected edges are contracted in $O(\text{sort}(E'_i))$ I/Os (using the algorithm in [12, 22, 25] or a simpler algorithm which we will include in the full version). After the contraction, the blocking value of a supervertex is set to be the minimum of the blocking values of the contracted vertices. The algorithm is correct as a simple induction argument can be be used to show that for every supervertex v the (contracted) edge sample contains all edges adjacent to v with weight smaller than the blocking value of v (i.e. the edges selected in the next phase belong to the MST). If in superphase i the blocking value of a supervertex v prevents us from selecting an edge for v to be included in the MST, then v must be the contraction of at least $\sqrt{N_i}$ vertices from V_i. This follows from the fact that the blocking value of v corresponds to the blocking value of some vertex u in V_i and v must span the $\lceil \sqrt{N_i} \rceil$ vertices adjacent to u in E'_i. If no blocking value prevents us from selecting an edges for v, then after $\lceil \log \sqrt{N_i} \rceil$ phases v must have size at least $2^{\log \sqrt{N_i}} = \sqrt{N_i}$. It follows that superphase i reduces the number of vertices by a factor of at least $\sqrt{N_i}$, i.e. the number of vertices after superphase i is at most $\frac{V_i}{\sqrt{N_i}} \leq \frac{V}{N_i\sqrt{N_i}} = \frac{V}{N_{i+1}}$ as claimed by the invariant.

Lemma 2. *Let $G = (V, E)$ be an undirected weighted graph. The MST problem on G can be reduced to the MST problem on a graph with at most $\frac{E}{B}$ vertices in $O(\mathrm{sort}(E) \cdot \log\log \frac{VB}{E})$ I/Os.*

3 Multi-way Planar Graph Separation

In this section, we show how to separate a planar graph G into $\Theta(\frac{N}{R})$ subgraphs with $O(R)$ vertices each and a set of $O(\mathrm{sort}(N))$ separator vertices using $O(\mathrm{sort}(N))$ I/Os.

Given a BFS tree T of G, Hutchinson *et al.* [19] showed how to compute a $O(\sqrt{N})$-separator for G in $O(\mathrm{sort}(N))$ I/Os. Their algorithm closely follows the algorithm by Lipton and Tarjan [23]: The BFS tree T has the property that no edge crosses two or more levels, and hence every level in T is a separator in G. The basic idea is to use the "middle" level ℓ_1 in T (the level containing the vertex with number $N/2$ in the BFS numbering) as the separator. Level ℓ_1 has the property that the total number of vertices on levels above ℓ_1, as well as in levels below ℓ_1, is less than $N/2$. The problem is that ℓ_1 might contain more than $O(\sqrt{N})$ vertices. However, there exists a level ℓ_0 above ℓ_1 and a level ℓ_2 below ℓ_1 with $O(\sqrt{N})$ vertices each, such that $\ell_2 - \ell_0 \leq \sqrt{N}$ (that is, ℓ_0 and ℓ_2 are not too far away from ℓ_1). Levels ℓ_0 and ℓ_2 divide G into three subgraphs G_0, G_1 and G_2 consisting of the vertices on the levels above ℓ_0, between ℓ_0 and ℓ_2 and below ℓ_2 respectively, with the property that G_0 and G_2 contain less than $N/2$ vertices and G_1 has a spanning tree of bounded height $\sqrt{N}$. Refer to Fig. 1 (a). It is easy to see that in order to find a separator for G it is enough to find a separator in G_1 [23]. Such a separator can be found using properties of the dual graph of G_1. The dual graph $G^\star = (V^\star, E^\star)$ of a planar graph G is a planar graph with a vertex for each face of G whose edges are in one-to-one correspondence with the edges of G. The dual graph $G^\star$ is obtained by placing a vertex in each face of G and connecting two faces f_i and f_j adjacent to a common edge $e = (u, v)$ of G with an edge (f_i, f_j) in $E^\star$. The edge (f_i, f_j) in $G^\star$ is called the dual edge of (u, v) in G. Let $E' \subseteq E$ be a subset of edges in G. It is well known that (V, E') is a spanning tree of G if and only if $(V^\star, (E - E')^\star)$ is a spanning tree in $G^\star$ [21]. Thus the edges in $(E - T)^\star$ form a spanning tree in $G^\star$ which we denote $T^\dagger$. An example is shown in Fig. 2(a). If T has bounded height $\sqrt{N}$ then every edge in $(E - T)$ (and therefore the corresponding edge in $(E - T)^\star$) determines a cycle in T with at most $2\sqrt{N}$ vertices. Assuming (without loss of generality) that G is triangulated, Lipton and Tarjan [23] proved that there exists an edge $e \in (E - T)$ such that the number of vertices inside and outside the cycle defined by e is $\leq 2N/3$, and showed how it can be computed efficiently using a bottom-up traversal of the dual tree $T^\dagger$. Hutchinson *et al.* [19] showed how to perform all these operations using $O(\mathrm{sort}(N))$ I/Os.

As discussed in the introduction, the $O(\mathrm{sort}(N))$ separator algorithm [19] can be used to develop a recursive $O(\log \frac{N}{R} \cdot \mathrm{sort}(N))$ multi-way separator algorithm in a straightforward way. The idea in our new $O(\text{sort}(N))$ algorithm is to obtain $O(\log_{M/B} \frac{N}{R})$ recursion depth by increasing the fan-out of the separation from 2 to $\frac{M}{B}$ and implement each step in $O(\frac{N}{B})$ I/Os. In order to divide the graph in $\frac{M}{B}$ subgraphs we use ideas similar to the ones used by Goodrich [18]. The general idea is the following: Instead of finding only one level cutting the graph in two halves, we find (roughly) $\frac{M}{B}$ levels which cut the graph in $O(\frac{N}{M/B})$-sized chunks. We then use these levels to find a set of levels with few vertices which divide G into subgraphs such that each subgraph is either of size $O(\frac{N}{M/B})$ or has a spanning tree of bounded height $O(\sqrt{R})$. We then subdivide the subgraphs with bounded height into graphs of size $O(R)$ using properties of the dual graph. In Section 3.2 we show how this can be done I/O-efficiently and prove the following lemma:

Lemma 3. *A graph G with a spanning tree T of height H can be divided into $\Theta(\frac{N}{R})$ subgraphs of size $O(R)$ each and $O(\frac{N}{R}H)$ separator vertices in total using $O(\text{sort}(N))$ I/Os.*

After subdividing the bounded height subgraphs we recursively subdivide the subgraphs of size $O(\frac{N}{M/B})$. In Section 3.1 we give the details in our algorithm and prove the following:

Theorem 2. *Let $G = (V, E)$ be a planar graph and T a breadth-first search tree for G. Furthermore assume $\exists\, \varepsilon > 0$ such that $M > B^{2+\varepsilon}$. For any $R = \Omega(M)$, G can be partitioned into $\Theta(\frac{N}{R})$ subgraphs G_i of size $O(R)$ each and a set of separator vertices S of size $O(\text{sort}(N))$ using $O(\text{sort}(N))$ I/Os.*

3.1 Separating Planar Graphs

In this section we prove Theorem 2 using Lemma 3. Let $L(i)$ be the total number of vertices on levels 0 through i of T and define the *starter levels* to be the levels i such that the interval $(L(i), L(i+1)]$ contains a multiple of $\lceil \frac{N}{X} \rceil$, for some $0 < X < N$. There are at most X starter levels and the number of vertices between consecutive starter levels is smaller than $\lceil \frac{N}{X} \rceil$.

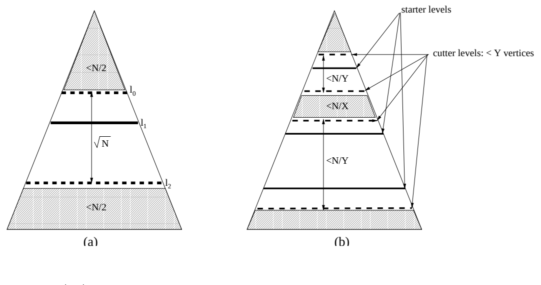

Fig. 1. (a) Illustration of the planar separator algorithm [23]; (b) Starter and cutter levels in T

Just like the ℓ_1 level in Lipton and Tarjan's algorithm [23], the starter levels divide G in subgraphs of "small" size. However, as previously, the starter levels can contain too many vertices. Therefore we consider the first level above each starter level, as well as the first level below each starter level containing at most Y vertices, for some $0 < Y < N$. We call these levels the *cutter levels.* The cutter levels divide G into $O(X)$ subgraphs G_i, consisting of the vertices between two consecutive cutter levels, with the property that if the two cutter levels defining G_i are within two (consecutive) starter levels then G_i has size $O(\frac{N}{X})$. If the two cutters defining G_i are not within two consecutive starter levels then G_i has a spanning tree of depth $O(\frac{N}{Y})$. Refer to Fig. 1 (b).

As mentioned, the idea in our algorithm is to apply Lemma 3 to the subgraphs of bounded height $O(\frac{N}{Y})$ and recursively separate the subgraphs of size $O(\frac{N}{X})$. By choosing $Y = \frac{N}{\sqrt{R}}$ each bounded height subgraph G_i of size N_i has height $\sqrt{R}$, and it can thus be separated into $\Theta(\frac{N_i}{R})$ subgraphs of size $O(R)$ and $O(\frac{N_i}{R} \cdot \sqrt{R}) = O(\frac{N}{\sqrt{R}})$ separator vertices using $O(\text{sort}(N_i))$ I/Os. Note that as we are not recursing on G_i (that is, we are not touching G_i again), the total cost of separating all such subgraphs over all levels of the recursion adds up to $O(\text{sort}(N))$ in total. The separator vertices are the vertices of the $O(X)$ cutter levels (each cutter level has at most $Y = \frac{N}{\sqrt{R}}$ vertices), the separator vertices resulting from applying Lemma 3 to the subgraphs of bounded height and the separator vertices resulted from the recursive calls. Thus the total number of separator vertices is given by $S(N) \leq X\frac{N}{\sqrt{R}} + \frac{N}{\sqrt{R}} + X \cdot S(\frac{N}{X})$. If we choose $X = (\frac{M}{B^2})^{1/4}$ and assume $M > B^{2+\varepsilon}$, for some $\varepsilon > 0$, it can be shown that $X\frac{N}{\sqrt{R}} = O(\frac{N}{B})$ and $\log_X \frac{N}{R} = O(\log_{M/B} \frac{N}{B})$, so that $S(N) = O(\text{sort}(N))$.

The only thing remaining to discuss is how to represent a subgraph G_i between two cutter levels c_i and c_{i+1} in the format needed in order to apply Lemma 3 or perform the recursive call. Both these steps require that a BFS tree is given along with the subgraph. The part of T included in G_i is not connected and thus it is not a BFS tree for G_i. However, we can easily produce such a tree by introducing a "fake" root v_i and connecting it with "fake" edges to all vertices on level c_{i+1}. Note that if T is given level-by-level this can easily be done for all the subgraphs in $O(\frac{N}{B})$ I/Os. The fake vertices and edges are marked so that they can be removed at the end of the algorithm. Details will appear in the full paper.

That our algorithm uses $O(\text{sort}(N))$ I/Os can be seen as follows. The preprocessing step of computing the BFS level for each vertex in T and sorting the edges of G by level can easily be performed in $O(\text{sort}(N))$ I/Os using standard techniques (such as list ranking and Euler tours) [12]. If we do not count the I/Os used to separate the subgraphs with bounded height, one recursion step can be performed in $O(\frac{N}{B})$ I/Os, and the recurrence for the number of I/Os used becomes $T(N) \leq \frac{N}{B} + X \cdot T(\frac{N}{X})$. Thus $T(N) = O(\text{sort}(N))$. As the total number of I/Os used to separate the subgraphs of bounded height is $O(\text{sort}(N))$, we have shown that our algorithm uses $O(\text{sort}(N))$ I/Os in total. This concludes the proof of Theorem 2.

So far we have only discussed the case $R = \Omega(M)$. If R is $o(M)$ then we can use Theorem 2 to separate G in subgraphs of size $O(M)$, then load each subgraph into main memory one at a time and apply Lipton and Tarjan planar separator algorithm [23] until all subgraphs have size $O(R)$. This results in $O(\frac{N}{\sqrt{R}})$ separator vertices. In some applications of the graph separation it is necessary to bound not only the total number of separators S, but also the number of separator vertices adjacent to any subgraph. This can be done as follows: For each subgraph which has $\Omega(\frac{S}{N/R})$ adjacent separator vertices mark the inner vertices as inactive and apply Theorem 2 until the resulting subgraphs have $O(\frac{S}{N/R})$ (active) vertices. Fredrickson [17] proves that this maintains the same bounds for the number of subgraphs and separators given that the graph has bounded degree. Details will appear in the full paper.

Corollary 1. *Let $G = (V, E)$ be a planar graph and T a breadth-first search tree for G. Furthermore assume $\exists\ \varepsilon > 0$ such that $M > B^{2+\varepsilon}$. Then G can be separated in $\Theta(\frac{N}{R})$ subgraphs of $O(R)$ vertices each and a set S of $O(\mathrm{sort}(N) + \frac{N}{\sqrt{R}})$ separator vertices using $O(\mathrm{sort}(N))$ I/Os.*

If G has bounded degree then the separation can be constructed such that each subgraph G_i is adjacent to $O(\frac{SR}{N})$ separator vertices.

3.2 Separating Planar Graphs of Bounded Height Spanning Tree

In this section describe how we can separate in $O(\mathrm{sort}(N))$ I/Os a planar graph $G = (V, E)$ with a spanning tree T of height H into $\Theta(\frac{N}{R})$ subgraphs of size $O(R)$ each and $O(\frac{N}{R}H)$ separator vertices.

Assume for simplicity that G is triangulated. (If this is not the case, we can triangulate it using $O(\mathrm{sort}(N))$ I/Os [19] and mark the added edges so that they can be removed at the end of the separation. Note that T remains a spanning tree after the triangulation). Let $G^\star$ be the dual of G and let $T^\dagger = (E - T)^\star$ be the spanning tree in $G^\star$. The spanning tree $T^\dagger$ can be computed from G and T in $O(\mathrm{sort}(N))$ I/Os using a face finding algorithm as in [19] and a few sorting steps. Each edge in $T^\dagger$ is the dual of an edge $e = (u, v)$ in $(E - T)$ and there exists a unique path from u to v in T; this path and e forms a cycle in G, and since T has bounded height H, the cycle contains at most $2H - 1$ vertices. Thus each edge in $T^\dagger$ determines a cycle of size $O(H)$ in G which separates G into the vertices inside the cycle and vertices outside the cycle. Refer to Fig. 2 (a). It can be shown that if e is the centroid edge of $T^\dagger$, then the number of vertices inside and outside the cycle is roughly the same [18].

The main idea in our algorithm is to find $O(\frac{N}{R})$ cycles which partition G into subgraphs of roughly equal size $O(R)$. In order to do so, we first discuss how to find $O(\frac{N}{R})$ edges in $T^\dagger$ such that their removal divides $T^\dagger$ into subtrees of roughly equal size $O(R)$. Then we show that the duals of these edges define $O(\frac{N}{R})$ cycles in G with the desired properties.

The decomposition of a tree into independent subtrees of approximately equal size was studied by Gazit *et al.* [27] in the context of parallel R-contractions. We review briefly their notations and results. Let $D = (V, E)$ be a tree with N vertices. The weight $W(v)$ of a vertex v in D is the number of vertices in the subtree rooted at v. A vertex v is called *R-critical* if v is not a leaf and $\lceil\frac{W(v)}{R}\rceil > \lceil\frac{W(v')}{R}\rceil$ for all children v' of v. Let $C \subset V$. Two edges e and e' of G are *C-equivalent* if there exists a path from e to e' that avoids the vertices C. The graphs induced by the equivalence classes of the C-equivalent edges are called the *bridges* of C. The *attachments* of a bridge I are the vertices of I that are also in C. The *R-bridges* of a tree D are the bridges of C, where C is the set of R-critical

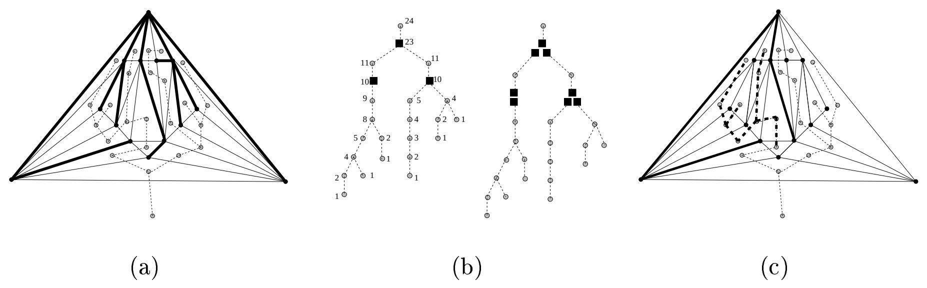

Fig. 2. (a) A triangulated graph G (solid lines), T (solid thick lines) and $T^\dagger$ (dotted lines). (b) The decomposition of $T^\dagger$ into its 10-bridges; square vertices are the attachments. (c) Subtree of $T^\dagger$ and the induced cycle in G.

vertices of D. An example of the decomposition of a tree into its R-bridges is shown in Fig. 2 (b). Gazit *et al.* [27] prove the following: (1) The number of R-critical vertices in a tree of size N is at most $\frac{2N}{R} - 1$. (2) The number of R-bridges in a tree with bounded degree d is at most $d(\frac{2N}{R} - 1)$. (3) The number of vertices of an R-bridge is at most $R + 1$. (4) If I is an R-bridge, then I can have at most two attachments.

As the basic step in the computation of the R-bridges of D is the computation of the weight of each vertex, it is easy to show how standard I/O-efficient algorithms can be used to compute the R-bridges in $O(\text{sort}(N))$ I/Os. If G is a triangulated graph, $T^\dagger$ is a binary tree, and thus it has at most $\frac{4N}{R}$ R-bridges. Each R-bridge defines two cycles in G determined by the two edges incident to the two attachments. One of these cycles will be inside the other and there are at most $R + 1$ faces inside the outer cycle but outside the inner cycle (the faces corresponding to the vertices in the R-bridge). Thus the R-bridges of $T^\dagger$ determine a separation of G into $\frac{4N}{R}$ subgraphs of at most R vertices adjacent to $O(\frac{N}{R}H)$ separator vertices in total. Given the R-bridges, the decomposition of G can be easily computed in $O(\text{sort}(N))$ I/Os and Lemma 3 follows.

4 Single Source Shortest Paths on Planar Graphs

In this section we show how to use our graph separation result to obtain an efficient SSSP algorithm for planar graphs with bounded degree.[3]

Consider separating a planar graph G into $\Theta(\frac{N}{R})$ subgraphs $G_i = (V_i, E_i)$ of $O(R)$ vertices each and a set S of separator vertices, such that each subgraph is adjacent to $O(\frac{SR}{N})$ separator vertices. We call the separator vertices adjacent to G_i the *boundary vertices* of G_i. Our algorithm relies on the following observation: Consider a shortest path $\delta(s,t)$ between two vertices s and t in G and let $\{s_0, s_1, ...\}$ denote its intersection with S. The portion of $\delta(s,t)$ between s_i and s_{i+1} is completely within some subgraph G_j and it must be the shortest path between s_i and s_{i+1} within G_j.

The main idea in our algorithm is to construct a new graph G^R by replacing each subgraph G_i with a complete graph on its boundary vertices. If the source vertex s is not a separator vertex, we also include s in G^R and connect it to the boundary vertices of the subgraph containing it. The graph

[3] Note that any graph can be transformed into a graph with each vertex having degree at most 3 using a simple transformation [17].

G^R has S vertices and $O(\frac{N}{R} \cdot (\frac{SR}{N})^2) = O(\frac{S^2 R}{N})$ edges. The weight of an edge in G^R is the length of the shortest path in G_i between the corresponding two boundary vertices. If $R = O(M)$ these weights can be computed as follows: We load each subgraph G_i into main memory together with its boundary vertices and use an internal memory all-pair-shortest-paths algorithm to compute the weights of the new edges between the boundary vertices of G_i, and write these edges to the disk. Since each separator vertex is a boundary vertex for at most $O(1)$ subgraphs (because of the bounded degree), we use at most $(\frac{N}{B} + S)$ I/Os to load all the subgraphs and their boundary vertices. As we use $O(\text{scan}(\frac{S^2 R}{N}))$ I/Os to write the new edges, it follows that G^R can be computed in $O(S + \text{scan}(\frac{S^2 R}{N}))$ I/Os in total. Using $S = O(\text{sort}(N) + \frac{N}{\sqrt{R}})$ (Corollary 1) and choosing $R = \frac{N^2}{\text{sort}^2(N)} = \frac{B^2}{\log^2_{M/B} N/B} < M$, this is $O(\text{sort}(N))$ I/Os.

Now assume we know how to compute the shortest paths from s to all separator vertices in $O(\text{sort}(N))$ I/Os. Using the observation mentioned above, we know that these paths are identical to the shortest paths in the original graph G. We can then compute the shortest paths from s to all the remaining vertices in G by loading each subgraph G_i and its boundary vertices in main memory, and using an internal memory algorithm to compute the shortest path from s to each vertex t in V_i using the formula $\delta(s,t) = \min_v\{\delta(s,v) + \delta_{G_i}(v,t)\}$, where v ranges over all boundary vertices of G_i. This takes $O(S + \text{scan}(N))$ I/Os, so the total number of I/Os used is $O(\text{sort}(N))$.

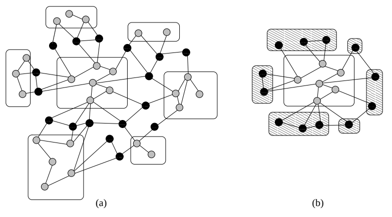

Fig. 3. (a) Separation of a graph into subgraphs (boxed) and separators (black); (b) a subgraph in the partition, its boundary vertices and boundary sets.

All that remains is to show how to solve the SSSP problem on the graph G^R with $S = O(\text{sort}(N))$ vertices and $O(\frac{S^2 R}{N}) = O(N)$ edges in $O(\text{sort}(N))$ I/Os. To do so we use a slightly modified version of Dijkstra's algorithm which avoids the use of a *decrease_key* priority queue operation. We want to avoid such an operation since the I/O bound of the best known external data structure with this operation is $O(\frac{\log_2 N}{B})$ [22], while priority queues with $O(\frac{\log_{M/B} N/B}{B})$ I/O bound are known if this operation is not supported [5, 9]. During the algorithm we maintain a list L of pairs of vertices of G^R and their distances. Initially all distances are ∞. We maintain the invariant that the distance of a vertex in L is identical to the distances stored in the priority queue controlling the algorithm. The algorithm repeatedly performs a *delete_min* operation on the priority queue to obtain the next vertex v to process; then the $O(\frac{SR}{N}) = O(\frac{B}{\log_{M/B} N/B})$ edges incident to v are loaded using $O(1)$ I/Os and the $O(\frac{SR}{N}) = O(\frac{B}{\log_{M/B} N/B})$ boundary vertices adjacent to v are determined. These vertices (and their current distances) are loaded from L using $O(\frac{B}{\log_{M/B} N/B})$ I/Os, and, without further I/Os we then compute which vertices need to have their distances updated. Finally, the new distances are written back to L and the corresponding updates are performed on the priority queue. Note that as we know the current distance of a vertex which needs to have its distance updated, we can perform the update in $O(\frac{\log_{M/B} N/B}{B})$ I/Os using a *delete* and an *insert* operation.

Our algorithm performs $O(N)$ operations on the priority queue using $O(\text{sort}(N))$ I/Os in total. It also uses $O(S) = O(\text{sort}(N))$ I/Os in total to load the neighbors of each vertex. Thus the I/O use is dominated by the $O(\frac{B}{\log_{M/B} N/B})$ I/Os used for each vertex to load its adjacent vertices from L. Since there are $O(\text{sort}(N))$ vertices, this sums up to $O(\frac{B}{\log_{M/B} N/B}) \cdot O(\text{sort}(N)) = O(N)$ I/Os in total.

In order to improve the I/O bound to $O(\text{sort}(N))$ we modify the algorithm, taking into account that there is some implicit adjacency between the boundary vertices. Let a *boundary set* be a maximal subset of boundary vertices such that all boundary vertices in the subset are adjacent to exactly the same subgraphs. An example is shown in Fig. 3 (b). Fredrickson [17] showed that the number of boundary sets is equal to the number of subgraphs $O(\frac{N}{R})$. We therefore modify our algorithm such that the vertices in the same boundary sets are stored consecutively in L. Otherwise the algorithm remains unmodified. When a vertex v is processed, the relevant boundary sets are determined and loaded from L as before. However, now we can think of the accesses as involving full boundary sets, as opposed to boundary vertices. Each boundary set is accessed $O(\frac{B}{\log_{M/B} \frac{N}{B}})$ times (once by each of its adjacent boundary vertices), and as there are $O(\frac{N}{R})$ boundary sets we use $O(\frac{B}{\log_{M/B} \frac{N}{B}} \cdot \frac{N}{R}) = O(\text{sort}(N))$ I/Os in total.

Theorem 3. *Let G be a bounded degree planar graph and T a BFS tree for G. Furthermore assume $\exists\, \varepsilon > 0$ such that $M > B^{2+\varepsilon}$. The SSSP problem on G can be solved in $O(\text{sort}(N))$ I/Os.*

References

1. J. Abello, A. L. Buchsbaum, and J. R. Westbrook. A functional approach to external graph algorithms. In *Proc. Annual European Symposium on Algorithms, LNCS 1461*, pages 332–343, 1998.
2. P. K. Agarwal, L. Arge, T. M. Murali, K. Varadarajan, and J. S. Vitter. I/O-efficient algorithms for contour line extraction and planar graph blocking. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 117–126, 1998.
3. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
4. ARC/INFO. *Understanding GIS—the ARC/INFO method.* ARC/INFO, 1993. Rev. 6 for workstations.
5. L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995. A complete version appears as BRICS technical report RS-96-28, University of Aarhus.
6. L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 1004*, pages 82–91, 1995. A complete version appears as BRICS technical report RS-96-29, University of Aarhus.
7. L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. In *Proc. Workshop on Algorithm Engineering and Experiments*, 2000.

8. O. Borůvka. O jistém problému minimálním. *Práca Moravské Přírodovědecké Společnosti*, 3:37–58, 1926.
9. G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 1432*, pages 107–118, 1998.
10. A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 859–860, 2000.
11. A. L. Buchsbaum and J. R. Westbrook. Maintaining hierarchical graph views. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 566–575, 2000.
12. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.
13. F. Chin, J. Lam, and I. Chen. Efficient parallel algorithms for some graph problems. *Communications of ACM*, 1982.
14. R. Cole and U. Vishkin. Approximate parallel scheduling. II. Applications to logarithmic-time optimal parallel graph algorithms. *Information and Computation*, 92(1):1–47, May 1991.
15. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1990.
16. E. Feuerstein and A. Marchetti-Spaccamela. Memory paging for connectivity and path problems in graphs. *LNCS*, 762:416–425, 1993.
17. G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal of Computing*, 16:1004–1022, 1987.
18. M. Goodrich. Planar separators and parallel polygon triangulation. *Journal of Computer and System Sciences*, 51(3):374–389, 1995.
19. D. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. In *Proc. 5th Annual Int. Conf. Computing and Combinatorics*, number 1627 in LNCS. Springer-Verlag, July 1999.
20. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, Reading MA, second edition, 1998.
21. D. Kozen. *The Design and Analysis of Algorithms.* Springer, Berlin, 1992.
22. V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. IEEE Symp. on Parallel and Distributed Processing*, pages 169–177, 1996.
23. R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal of Applied Math.*, 36:177–189, 1979.
24. A. Maheshwari and N. Zeh. External memory algorithms for outerplanar graphs. Manuscript, 1999.
25. K. Munagala and A. Ranade. I/O-complexity of graph algorithm. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 687–694, 1999.
26. M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.
27. J. H. Reif, editor. *Synthesis of Parallel Algorithms*, chapter 3, pages 115–194. Morgan Kaufmann, 1993.
28. R. E. Tarjan. *Data structures and network algorithms.* SIAM, Philadelphia, 1983.
29. J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intellegence*, 3:331–360, 1991.
30. J. S. Vitter. External memory algorithms (invited tutorial). In *Proc. of the 1998 ACM Symposium on Principles of Database Systems*, pages 119–128, 1998.