A Parallel Algorithm for Approximating the Minimum Cycle Cover

Philip Klein*
Aiken Computation Laboratory
Harvard University
Cambridge, MA

Clifford Stein†
Laboratory for Computer Science
MIT
Cambridge, MA

Abstract

We address the problem of approximating a *minimum cycle cover* in parallel. We give the first efficient parallel algorithm for finding an approximation to a *minimum cycle cover*. Our algorithm finds a cycle cover whose size is within a factor of $O(1 + \frac{n \log n}{m+n})$ of the minimum sized cover using $O(\log^2 n)$ time on $(m+n)/\log n$ processors.

1 Introduction

In this paper we design a parallel approximation algorithm for the problem of finding a *minimum cycle cover*. A cycle cover is a set of cycles such that every edge in the graph appears in at least one cycle. We restrict our attention to 2-edge connected (bridgeless) graphs, as any graph with a bridge does not have a cycle cover. In applications such as the analysis of irrigation systems by the Hardy Cross method [5] and the analysis of electrical circuits, it is important to find a small cycle cover. Finding a minimum cover—one minimizing the sum of the sizes of the cycles—is conjectured to be NP-complete [7]. We give the first efficient parallel approximation algorithm for this problem, as we can find an $O(1 + \frac{n \log n}{m+n})$ approximation to the minimum cycle cover in $O(\log^2 n)$ time on $(m+n)/\log n$ processors.

Additionally, our techniques yield a useful sequential algorithm. The sequential algorithm that finds the smallest cycle cover is that of Alon and Tarsi [1]; their

*Research supported by ONR grant N00014-88-K-0243 and DARPA grant N00039-88-C0113 at Harvard University. This author's current address is Computer Science Department, Brown University, Providence, RI

†Research supported by a graduate fellowship from GE. Additional support provided by Air Force Contract AFOSR-86-0078, and by an NSF PYI awarded to David Shmoys, with matching funds from IBM, Sun Microsystems, and UPS.

algorithm guarantees a constant factor approximation. However, their algorithm requires $O(m + n^2)$ time. Thus by sacrificing an $O(1 + \frac{n \log n}{m+n})$ factor in the size of the cover, we obtain a faster algorithm, one that runs in $O(m + n \log n)$ time . Note that for non-sparse graphs ($m > n \log n$), our techniques yield a cover whose size is within a constant factor of optimal. Further, for all classes of graphs, our algorithm is faster than any of the previous algorithms for finding a cycle cover [1, 7, 8].

The basis for our approximation algorithm is a routine to find a maximal set of edge-disjoint cycles in an undirected graph, given in [12]. Thus we demonstrate the utility of this technique in parallel combinatorial optimization. Historically, many sequential graph algorithms have relied on decomposing a graph with a breadth-first search or a depth-first search. However, no fast and efficient parallel algorithm is known for either of these problems, i.e. no polylogarithmic-time algorithm is known that has a processor-time product even close to linear in the number of nodes and edges in the input graph. Our work is part of an effort by researchers to design fast and efficient parallel decompositions more suitable for use in parallel computation. Other such decompositions that have proven fruitful include ear decompositions [14, 15], and Euler tours [20, 2, 13, 6].

2 Finding a Maximal Set of Edge Disjoint Cycles

We begin by reviewing an algorithm to find a maximal set of edge-disjoint cycles in $O(\log n)$ time on $(m + n)/\log n$ processors. A slight variation on this algorithm will be an important part of our algorithm to find a minimum cycle cover. For details, the reader is referred to [12].

2.1 Preliminaries

Let $G = (V, E)$ be an n-node undirected graph with node set V and edge set $E = \{e_1, \ldots, e_m\}$. Let $\{0, 1\}^E$ denote the m-dimensional vector space over $GF(2)$. Each subgraph H of G corresponds to a vector $\mu(H) = (\mu_1, \ldots, \mu_m)$, where $\mu_i = 1$ if edge e_i appears in H, and $\mu_i = 0$ otherwise. An *even-degree subgraph* of G is a subgraph in which every node has even degree. Every connected component of an even-degree subgraph is Eulerian; hence, the edges of such a subgraph can be decomposed into cycles. Furthermore, it is well-known (and easy to see) that the vectors corresponding to even-degree subgraphs of G form a vector subspace of $\{0, 1\}^E$, called the *cycle space* of G and denoted $\mathcal{C}(G)$. That is, for two even-degree subgraphs H_1 and H_2 of G, the elementwise mod 2 sum of the vectors $\mu(H_1)$ and $\mu(H_2)$ is again a vector $\mu(H)$

corresponding to an even-degree subgraph H.

Let F be a spanning forest of G, i.e. let F be a collection of spanning trees, one for each connected component of G. Let $\hat{F}$ be a rooted version of F; that is, $\hat{F}$ is obtained from F by choosing a root within each tree of F. For each node v, there is a unique simple path $P(v)$ from v to the root of the tree containing v. For two nodes v and w belonging to the same tree, the *lowest common ancestor* of v and w, denoted $lca(v, w)$, is the first node common to $P(v)$ and $P(w)$.

An edge $e = (v, w)$ of G not appearing in F is called a *cycle-edge* (with respect to F) because $F \cup \{e\}$ contains a unique simple cycle, denoted $C(e)$. In particular, the cycle $C(e)$ consists of e, together with the paths from v and w to their lowest common ancestor in $\hat{F}$. Using the $GF(2)$ vector notation, we can write the cycle as

$$\mu(C(e)) = \mu(e) + \mu(P(v)) + \mu(P(w)). \tag{1}$$

because the portions of $P(v)$ and $P(w)$ from $lca(v, w)$ to the root coincide and cancel each other. It is well known that this collection of cycle vectors $\{\mu(C(e)) | e \notin F\}$ forms a *cycle basis*, i.e. a set of linearly independent vectors which span the cycle space [3]. We will refer to this particular basis as $\mathcal{B}$.

2.2 Algorithm *Maximal Cycles*

Consider the even-degree subgraph H obtained by adding together mod 2 all the cycles $C(e) \in \mathcal{B}$. That is, H is defined by

$$\mu(H) = \sum_{e \notin F} \mu(C(e)) \tag{2}$$

where the sum is elementwise mod 2. Since each edge e not in F occurs exactly once in the sum, e is in the subgraph H. Thus H contains all non-forest edges and possibly some forest edges. However, all edges not in H must be in F, so the set of edges not in H form an acyclic graph. It follows that when the edges of H are removed from G, the remaining edges form an acyclic subgraph. Moreover, H can be decomposed into edge-disjoint cycles. These observations suggest the following algorithm, *Maximal Cycles*, for finding a maximal set of edge-disjoint cycles in G:

Step 1: Choose a rooted spanning forest $\hat{F}$ of G.

Step 2: Determine the subgraph H of G by (2).

Step 3: Decompose H into edge-disjoint cycles.

In [12], we show how to efficiently implement the above algorithm, yielding the following theorem:

Theorem 2.1 *Algorithm* Maximal Cycles *finds a maximal set of edge disjoint cycles in an undirected graph in $O(\log n)$ time on $(n+m)/\log n$ processors.*

3 Approximating a Cycle Cover

In this section, we develop new algorithms for approximating the *minimum cycle cover*. A cycle cover of a graph is a set of cycles in which each edge of the graph is in at least one cycle. The size of a cycle cover is the sum of the sizes of its constituent cycles, where the size of a cycle is defined to be the number of edges in the cycle. The minimum cycle cover is the cycle cover of smallest cardinality. We give a series of parallel algorithms, each of which finds a smaller cover than the previous one. In Section 3.1, we give the history of the problem. In Section 3.2, we give a more involved algorithm that finds a cover of size $O(m \log n)$, and in Section 3.3, we improve the size of the cover to be $O(m + n \log n)$. We conclude by observing that these techniques also yield an efficient sequential algorithm.

3.1 Preliminaries and Background

Let $\mathcal{C} = \{C_1, \ldots, C_k\}$ be a set of simple cycles in the undirected graph $G = (V, E)$, and let $E(\mathcal{C})$ be the set of edges contained in $\mathcal{C}$. More generally, we denote the set of edges in some subgraph S by $E(S)$. We say that $\mathcal{C}$ is a *cycle cover* of the graph $G = (V, E)$ if $E(\mathcal{C}) = E$, i.e. every edge is in at least one cycle in the set $\mathcal{C}$. We define the size of a cycle cover to be the total number of edges in the cycles that constitute that cover, i.e. $|\mathcal{C}| = \sum_{C_i \in \mathcal{C}} |E(C_i)|$. The *minimum cycle cover* is the cycle cover for which $|\mathcal{C}|$ is minimized. An edge of a graph is a *bridge* if no cycle in the graph contains that edge. Clearly, a graph has a cycle cover if and only if the graph is bridgeless.

Sequential algorithms for finding a cycle cover have been developed with two goals in mind. The first goal is to find a cover of small size, and the second is to get an algorithm that runs quickly. The first algorithm for this problem, by Itai and Rodeh [8], finds a cover of size $O(m + n \log n)$ in $O(n^3)$ time. Subsequently, Itai, Lipton, Papadimitriou and Rodeh [7] showed that every graph has a cover of size $\min\{3m - 6, m + 6n - 7\}$ and that this cover can be found in $O(n^2)$ time. This result relies on a result of Jaeger [9] that shows that every bridgeless graph has a nowhere zero flow modulo 8, and results of Tarjan [19] and Shiloach [18] that find edge-disjoint branchings. They also conjecture that finding the minimum cycle cover is NP-complete. Alon and Tarsi [1] have developed an algorithm that finds a smaller cover, one of size at most $\min\{\frac{5}{3}m, m + \frac{7}{3}n - \frac{7}{3}\}$, and runs in $O(m+n^2)$ time. This result relies on a proof by Seymour [17] that every bridgeless graph has a nowhere zero flow modulo 6. Alon and Tarsi also note that a certain graph called the Petersen graph [8, 7] has 15 edges and no cycle cover of size less than 21. This graph can be generalized to show that there exists an infinite family of graphs of m edges that have a minimum cycle cover of size at least $\frac{7}{5}m$.

In this section, we consider the problem of finding a cycle cover in parallel. We first note that all the sequential algorithms mentioned above rely on edge-disjoint branchings and nowhere zero flows. All algorithms known to us for these problems require the computation of $\Omega(n)$ maximum flows on graphs with polynomial bounded capacities. Even if this sequence of computations could be efficiently parallelized, the best known NC algorithm for computing one maximum flow in a graph with polynomial bounded capacities uses many processors and randomness [11].

Thus, we focus on a different strategy that is based on using the algorithm *Maximal Cycles* as a subroutine. First observe that the output of this algorithm is a set of cycles $\mathcal{C}$ such that $m - n + 1 \leq |E(\mathcal{C})| \leq m$. Thus, we already know how to find a set of cycles that cover all but $n-1$ or fewer of the edges. We could then cover each of the remaining edges with a cycle using $(m+n)/\log n$ processors per edge in $O(\log n)$ time, yielding a cycle cover of size $m + n(n-1)$ using $O(\log n)$ time on $n(m+n)$ processors. This gives some intuition about the relationship between maximal sets of edge-disjoint cycles and cycle covers. However the size of the cover and the number of processors used are too large to be of practical interest; we will focus on finding a cover of smaller size.

3.2 Finding a Cover of Size $O(m \log n)$

In this section we will use the algorithm *Maximal Cycles* to develop an algorithm that finds a cycle cover of size $O(m \log n)$. Our algorithm will proceed in iterations; in each iteration we will use the algorithm *Maximal Cycles* to choose a set of cycles to add to our collection $\mathcal{C}$. We will take advantage of the freedom to choose which edges to place in the spanning tree. In general, during iteration i we will prefer to put in the spanning tree edges that were put into $\mathcal{C}$ in some iteration previous to i. Thus, we will force as many uncovered edges as possible to be non-tree edges. This means that they will be included in some cycle during iteration i and hence added to $\mathcal{C}$.

Input: Bridgeless graph $G = (V, E)$.
Output: $\mathcal{C}$, a cycle cover of size $O(m \log n)$.

1 Initialize $\mathcal{C}$ to empty.
2 While $E(\mathcal{C}) \neq E$
3   Let $G_{\mathcal{C}} = (V, E(\mathcal{C}))$.
4   Find F, a spanning forest of $G_{\mathcal{C}}$.
5   Find T, a spanning tree of G containing all edges in F.
6   Find $\Delta\mathcal{C} = \{C_1, \ldots, C_k\}$, a maximal set of edge-disjoint cycles in G covering all edges not in T.
7   Add the cycles in $\Delta\mathcal{C}$ to $\mathcal{C}$.

Figure 1: Algorithm *Cycle Cover*

To achieve this, we will first find a spanning forest of the graph induced by $E(\mathcal{C})$, the edges already covered. Next, we extend this to a spanning tree of G. Given this tree, we use Steps 2 and 3 of the algorithm *Maximal Cycles* to find a maximal set of edge-disjoint cycles in G, which we then add to $\mathcal{C}$. The algorithm appears in Figure 1. Informally, this strategy achieves our goal of putting as few uncovered edges as possible in the spanning tree; we will now proceed to show this more rigorously.

The key to the analysis of the algorithm is to show that the number of iterations is $O(\log n)$. At the beginning of each iteration, we have a graph G in which a set $E(\mathcal{C})$ of edges are covered. We would like to be able to show that in each iteration, a constant fraction of the uncovered edges become covered. As we will show, if all nodes have degree three or more, this is true. However, if a graph has nodes of degree two, it is not necessarily the case that a constant fraction of the uncovered edges become covered. However, by defining progress in terms of an auxiliary graph in which every node has degree three or more, we obtain the desired result.

We derive an auxiliary graph $H = H(G, E(\mathcal{C}))$ from G and the covered edges such that each iteration reduces the number of edges of H by a constant fraction. To obtain H from G, first contract all the covered edges $E(\mathcal{C})$, then repeatedly splice out all nodes of degree two until all nodes in the graph have degree three or more. (To "splice out" a node of degree two is to contract one of its incident edges.)

Lemma 3.1 *Let $\mathcal{C}$ be the cycle cover at the beginning of some iteration of algorithm* Cycle Cover*, and let $\Delta\mathcal{C}$ be the collection of edge-disjoint cycles found in step 6. Then either $H(G, E(\mathcal{C} \cup \Delta\mathcal{C}))$ has at most two-thirds the edges of $H(G, E(\mathcal{C}))$, or the algorithm terminates immediately.*

Proof: Let T be the spanning tree of G and $H = H(G, E(\mathcal{C}))$ be the auxiliary graph defined above. Then, T_H, the spanning tree of H can be obtained from the spanning tree T of G by contraction: contract each edge of T that was contracted in obtaining H from G. Every non-tree edge in H with respect to T_H is a non-tree edge in G with respect to T, so the edges covered by the cycle collection $\Delta\mathcal{C}$ found in step 6 include all non-tree edges of H. There are two cases to consider:

Case 1: (H has at least one edge.) Let n_H be the number of nodes in H. Since every node of H has degree at least three, H has at least $\frac{3}{2}n_H$ edges. The number of tree-edges in H is $n_H - 1$, hence the number of nontree edges of H is more than one-third the number of edges of H. Thus the graph H' obtained from H by contracting nontree edges has fewer than two-thirds the edges of H. But the graph $H(G, E(\mathcal{C} \cup \Delta\mathcal{C}))$ is obtainable from H' by contractions, and so has no more edges than H'.

Case 2: (H has no edges) Since contraction preserves connectivity H must consist of a single isolated node. Then the graph G', obtained from G by contracting edges covered by $\mathcal{C}$, is connected with degree at most two, and no nodes of degree one. Hence is a simple cycle (or a single isolated node). In this case, we claim that the collection $\Delta\mathcal{C}$ of step 6 covers all as-yet-uncovered edges of G, and hence that the algorithm terminates. To prove the claim, simply contract the edges of $\Delta\mathcal{C}$ that are already in $\mathcal{C}$. The resulting cycle collection $\Delta\mathcal{C}'$ is a subgraph of G'; since G' consists of a simple cycle, $\Delta\mathcal{C}'$ must include every edge of G'. ■

Given this lemma, we can prove the following theorem:

Theorem 3.2 *Algorithm* Cycle Cover *finds a cycle cover of size* $O(m \log n)$ *in* $O(\log^2 n)$ *time using* $(m+n)/\log n$ *processors.*

Proof: Each iteration of the while loop finds a cycle cover of size at most m. By Lemma 3.1, there are $O(\log n)$ iterations and each iteration is the algorithm *Maximal Cycles* with some restrictions placed on the choice of the spanning tree. By looking at the spanning tree algorithm of [10], we see that this only makes the problem easier and hence adds no additional resources. ■

We note that in practice, we would change Step 7 to include a cycle $\Delta\mathcal{C}$ only if it contained some edge that was not already in $\mathcal{C}$. This could be checked in $O(\log n)$ time on $(m+n)/\log n$ processors using pointer jumping [4]. Also, we could replace Steps 3, 4, and 5 with the computation of a minimum spanning tree of G with the edges in $E(\mathcal{C})$ weighted with 0 and the rest of the edges weighted with 1.

3.3 Finding a Cover of Size $O(m + n\log n)$

In this section, we show how to decrease the size of the cycle cover from $O(m \log n)$ to $O(m + n \log n)$ using no additional resources. This algorithm is closely related

Input: A bridgeless graph G and a rooted spanning tree $\hat{T}$.
Output: A bridgeless graph $B = (V, E_B)$ s.t. $\hat{T} \subseteq B$ and $|E_B| \leq 2n$.

$\forall v \in V$, compute $level(v)$, the distance from v to the root of $\hat{T}$.

$\forall (v, w) \notin \hat{T}$, compute $lca(v, w)$.

$\forall v \in V$, let $N(v)$ be the non-tree neighbor w minimizing $level(lca(v, w))$ or $\emptyset$ if v has no non-tree neighbors.

Let $E_B = T \cup \{(v, N(v)) \ : \ v \in V, N(v) \neq \emptyset\}$.

Figure 2: **Algorithm** *Sparse Bridgeless Subgraph*

to the ear decomposition algorithm of Lovasz [14]. Observe that the first iteration of the algorithm finds a cover of size no less than $m - n + 1$. Thus the number of edges not in $\mathcal{C}$ is at most $n - 1$. Assume that we can form a bridgeless graph B that contains all the uncovered edges, and has at most $2n$ edges. We can run the algorithm *Cycle Cover* on B and obtain a cover of size $O(n \log n)$. Then we can combine this cover with the set of cycles obtained from the first iteration of the algorithm on G, and obtain a cycle cover of size $O(m + n \log n)$, using no additional resources. It is known how to find such a graph B in linear time sequentially [7], but this requires using depth-first search. We present a parallel algorithm that does not use depth-first search and finds a graph B in $O(\log n)$ time using $(m + n)/\log n$ processors.

First, for each node v, we compute $level(v)$, its distance from the root in some rooted spanning tree $\hat{T}$. Then, for each non-tree edge (v, w), we compute $lca(v, w)$. Finally, we choose the edges of B by including the spanning tree $\hat{T}$, and, for each node v which has at least one non-tree neighbor, the edge (v, w), where w is the node that minimizes $level(lca(v, w))$. This algorithm, *Sparse Bridgeless Subgraph*, appears in Figure 2.

Lemma 3.3 *Given a bridgeless graph G and a spanning tree T, algorithm* Sparse Bridgeless Subgraph *computes a bridgeless graph $B = (V, E_B)$ s.t. $T \subseteq B$ and $|E_B| \leq 2n - 1$.*

Proof: Since E_B contains the $n - 1$ tree edges and at most n additional edges (one per node), $|E_B| \leq 2n - 1$. Since B contains a spanning tree, B is connected. Now we will argue that B is bridgeless. Assume, for the sake of contradiction, that B contains a bridge. Further, since all non-tree edges are trivially not bridges, we assume, without loss of generality, that the bridge is a tree edge (x, y) with x the parent of y. Because (x, y) is a bridge, there is no non-tree edge from a (not necessarily proper) descendant of y to a non-descendant of y. Thus, for all descendants v of y, and all edges $(v, w) \in E_B$, $level(lca(v, w)) \geq level(y)$. But recall that we chose the non-tree edges of B to be the ones which minimized $level(lca(v, w))$ in G. Thus, in G, for every descendant v of y, it is the case that $level(lca(v, w)) \geq level(y)$. In other words, in G, there is no non-tree edge from a descendant of y to a non-descendant of y. Thus, (x, y) is a bridge in G. However, this contradicts our assumption that G is bridgeless. ■

Combining Lemma 3.3 with Theorem 3.2 we get the main result of this section.

Theorem 3.4 *Combining algorithms* Cycle Cover *and* Sparse Bridgeless Subgraph *yields an algorithm that finds a cycle cover of size $O(m + n \log n)$ using $O(\log^2 n)$ time on $(m + n)/\log n$ processors .*

Proof: The bound follows from the previous results and the results of [16] that show how to compute *lca* and *level* in the stated time bounds. ■

Our parallel algorithm translates into an efficient sequential algorithm.

Theorem 3.5 *A sequential implementation of algorithms* Cycle Cover *and* Sparse Bridgeless Subgraph *finds a cycle cover of size $O(m + n \log n)$ in sequential time $O(m + n \log n)$.*

Proof: The first maximal set of edge-disjoint cycles can be computed in $O(m+n)$ time by algorithm *Maximal Cycles*. We then find a sparse bridgeless subgraph in $O(n+m)$ time. Finally we run algorithm *Cycle Cover* on the sparse graph in $O(n \log n)$ time. ■

The cover we find is within a factor of $O(1 + \frac{n \log n}{m+n})$ of optimal. Thus for graphs with $m > n \log n$, the size of the cover is within constant factor of optimal, and the algorithm is faster than any of the previous algorithms.

4 Concluding Remarks and Open Problems

We have given algorithms which find a $O(1 + \frac{n \log n}{m+n})$ approximation to the minimum cycle cover. Previous sequential algorithms for this problems used techniques that were not easily parallelizable. However, by focusing a subproblem which involves finding a *maximal* sets of objects as opposed to a subproblem which involves finding a *maximum* set of objects, we were able to develop an algorithm that is nearly optimal in its use of parallelism.

Our algorithm uses fewer resources than the known sequential algorithms and gets a slightly worse approximation. It is an interesting open problem to develop a constant factor approximation algorithm which uses n^2 processors and $O(\log^k n)$ time.

Acknowledgments

We are grateful to David Shmoys, Éva Tardos, Joel Wein, and James Park for helpful discussions.

References

[1] N. Alon and M. Tarsi. Covering multigraphs by simple circuits. *SIAM Journal of Algebraic and Discrete Methods*, 6:345–350, 1985.

[2] B. Awerbach, A. Israeli, and Y. Shiloach. Finding euler circuits in logarithmic parallel time. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 249–257, 1984.

[3] C. Berge. *Graphs and hypergraphs.* North Holland Mathematical library, 1979.

[4] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree, and graph problems. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 478–491, 1986.

[5] H. Cross. Analysis of flow in networks of conduits of conductors. Bulletin 286, University of Illinois Engineering Experimental Station, Urbana, Ill., 1936.

[6] H. Gabow. Using euler partitions to edge-color bipartite multigraphs. *International Journal of Computing and Information Science*, 5:345–355, 1976.

[7] A. Itai, R. J. Lipton, C.H. Papadimitriou, and M. Rodeh. Covering graphs by simple circuits. *SIAM Journal on Computing*, 10(4):746–750, 1981.

[8] A. Itai and M. Rodeh. Covering a graph by circuits. In *Proceeding of the ICALP Conference*, Udine, 1978.

[9] F. Jaeger. On nowhere-zero flow in multigraphs. In *Proceedings of the Fifth British Combinatorial Conference*, pages 373–378, 1975.

[10] H. Jung. An optimal parallel algorithm for computing connected components in a graph. Preprint, Humboldt University, Berlin, German Democratic Republic, 1989.

[11] R. Karp, E. Upfal, and A. Wigderson. Constructing a perfect matching is in random *NC*. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 22–32, 1985.

[12] P. Klein and C. Stein. A parallel algorithm for eliminating cycles in undirected graphs. *Information Processing Letters*, 34(6):307–312, May 1990.

[13] G. F. Lev, N. Pippenger, and L. G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computers*, C-30:93–100, 1981.

[14] L. Lovász. Computing ears and branchings in parallel. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pages 464–467, 1985.

[15] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (EDS) and st-numbering in graphs. In *VLSI algorithms and architectures, Lecture notes in computer science 227*, pages 34–45. Springer-Verlag, 1986.

[16] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. In *Agaean Workshop on Computing*, pages 111–123, 1988. Published as Lecture Notes in Computer Science 319, Springer-Verlag.

[17] P.D. Seymour. Nowhere-zero 6 flows. *Journal of Combinatorial Theory B*, 30:130–135, 1981.

[18] Y. Shiloach. Edge-disjoint branching in directed multigraphs. *Information Processing Letters*, 8:24–27, 1979.

[19] R. E. Tarjan. A good algorithm for edge-disjoint branchings. *Information Processing Letters*, 3:51–53, 1975.

[20] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 12–20, 1984.