Using Cycles and Scaling in Parallel Algorithms

by

Clifford Stein

B.S.E., Computer Science
Princeton University
(1987)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology
August 1989

Signature of Author ______________________________
Department of Electrical Engineering and Computer Science
August 18, 1989

Certified by ______________________________
David Shmoys
Associate Professor of Mathematics
Thesis Supervisor

Accepted by ______________________________
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Using Cycles and Scaling in Parallel Algorithms

by

Clifford Stein

Submitted to the
Department of Electrical Engineering and Computer Science
on August 18, 1989
in partial fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

We introduce the technique of decomposing an undirected graph by finding a *maximal set of edge-disjoint cycles.* We give a parallel algorithm to find this decomposition in $O(\log n)$ time on $(m+n)/\log n$ processors. We then use this decomposition to 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. We also generalize these algorithms to weighted graphs with running times that are a factor of $O(\log C)$ slower than their unweighted counterparts, where C is the largest weight in the graph. Finally, we show how to use *scaling* to develop parallel algorithms for the assignment problem in which the number of processors used is independent of the magnitude of the edge costs. This leads to algorithms for the assignment problem that do less *work* than any known RNC algorithms for this problem.

Portions of this thesis are joint work with Philip Klein.

Thesis Supervisor: David Shmoys

Title: Associate Professor of Mathematics

Keywords: parallel algorithms, undirected graphs, cycles, Euler tour, cycle cover, scaling, matching, assignment problem.

The author was supported in part by a graduate fellowship from GE, 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.

Acknowledgments

I am deeply greatful to David Shmoys for advising this thesis. His support, encouragement, guidance, and ability to point me in the right direction were invaluable. He has been extremely generous with his time and ideas and was always willing to listen.

I am very lucky to have the privilege of working with Philip Klein. I have learned a great deal about research, writing, and scholarship from Philip. I thank him for allowing me to include our joint work in Chapter 2 and for helping provide the proof of Lemma 3.2.1.

I thank Margaret Tuttle and David Williamson for carefully reading a draft of this thesis. I also thank Joel Wein, James Park, Éva Tardos, Jim Orlin, Mark Hansen and Bruce Maggs for many helpful discussions, and Tom Cormen for help with Figure 2.2.

I am greatful to my parents for their constant love and support. I am also greatful to Rebecca Ivry for her support and understanding and for always being there for me. Finally, I thank my officemates Tom Cormen, Bruce Maggs, and Sally Goldman for making our office an interesting and comfortable place to be.

Contents

To May Stein, who would have been proud.

Chapter 1

Introduction

Many graph-theoretic problems that are "easy" to solve sequentially turn out to be much more difficult to solve in parallel. For example, most linear-time sequential graph algorithms rely 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. Thus, in order to design fast and efficient parallel algorithms for graph theoretic problems, we must consider new, or at least different, types of graph decompositions and algorithmic techniques. Examples of such decompositions that have proven fruitful include ear decompositions [35, 36], and Euler tours [44, 8, 34, 17]; examples of useful general techniques include divide and conquer [32, 28] and dynamic programming [1]. The novel use of these decomposition and algorithmic techniques has led to efficient parallel algorithms for a variety of problems, and in some cases to improved sequential algorithms as well. In this thesis we introduce a new technique of decomposing a graph into a *maximal set of edge-disjoint cycles* and show how an old algorithmic technique, *scaling*, can be used to achieve improved parallel algorithms for the assignment problem.

In Chapter 2, we introduce the technique of decomposing a graph by finding a *maximal set of edge-disjoint cycles.* Given a graph, we would like to identify a collection of edge-disjoint cycles whose removal renders the graph acyclic. We give a parallel algorithm that solves this problem for n-node, m-edge undirected graphs in $O(\log n)$ time using $(m+n)/\log n$ processors of a concurrent-read concurrentwrite parallel random access machine (CRCW PRAM) [31]. Since the problem requires $\Omega(n+m)$ operations in the worst case, our algorithm is optimal in its use of parallelism.

This technique is related to the *Euler tour* technique. In 1736, Euler posed the first graph theoretic problem, known as the Königsberg Bridge Problem. This problem can be phrased in graph-theoretic terms as: given an undirected graph G, find a cycle that covers every edge in the graph exactly once. A cycle of this form is called an *Eulerian cycle* and a graph that contains such a cycle is called an *Eulerian graph.* Euler showed that a graph is Eulerian if and only if the graph is connected and every node in the graph has even degree. Moreover, there is a linear-time sequential algorithm which, given an Eulerian graph, finds an Eulerian cycle. Awerbach, Israeli, and Shiloach [8] have given a parallel algorithm to solve this problem in $O(\log n)$ time using $n+m$ processors, for graphs with n nodes and m edges.

As this technique has proven useful for Eulerian graphs, there has been work on approximating Eulerian tours in non-Eulerian graphs. The standard technique is to convert a non-Eulerian graph into an Eulerian graph by adding a new node v', and an edge between every odd degree node and v'. However, this technique does not always prove to be useful algorithmically. Finding a maximal set of edge-disjoint cycles can be viewed as an alternative method of approximating an Eulerian tour, as it defines a *maximal Eulerian subgraph.*

We also introduce a generalization of our algorithm that handles integer-weighted undirected graphs as well. We think of the edge weights as edge multiplicities and find a set of cycles with multiplicities, whose removal renders the graph acyclic. The algorithm takes $O(\log n \log C)$ time on $(m+n)/\log n$ processors of a CRCW PRAM, where C is the largest edge weight in the graph. We hope that this generalization will be of use in solving network optimization problems, as the weights allow us to model the notions of flow and capacity.

In Chapter 3, we demonstrate the utility of this technique by using it to find a *cycle cover* of a biconnected graph. A cycle cover is a set of cycles such that every edge in the graph appears in at least one cycle. In applications such as the analysis of irrigation systems by the Hardy Cross method [13] and the analysis of electrical circuits, it is important to find a small cycle cover. Finding a minimum cover—one using the fewest possible edges—is conjectured to be NP-complete [25]. 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. We also generalize this algorithm to multigraphs using $O(\log^2 n \log C)$ 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 [6]; their 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 [6, 25, 26].

In Chapter 4, we demonstrate how *scaling* can be used to substantially reduce the amount of work needed to find a minimum perfect matching in a bipartite graph. Karp, Upfal, and Wigderson [30], and Mulmuley, Vazirani, and Vazirani [38], have recently developed randomized NC (RNC)[1] parallel algorithms for the minimum perfect matching problem, assuming that the input is given in unary. For both of these algorithms, the number of processors needed is proportional to the *magnitude* of the largest edge cost. We show how to convert these algorithms into algorithms that use a number of processors that is *independent of the magnitude of the largest edge cost*, provided that the graph is bipartite. As a tradeoff, we get an increase in the time spent that is proportional to the logarithm of the magnitude of the largest number. If $C = \Omega(n^{1+\epsilon})$, we get algorithms that do less work, where work is the product of the number of processors used and the time spent. Assuming similarity[2], our algorithms are in RNC.

We achieve these results by using *scaling*, which reduces the problem of finding

[1] NC is the class of algorithms that, on input of size n, use n^{k_1} processors and $O(\log^{k_2} n)$ time, for some constants k_1 and k_2. RNC algorithms are NC algorithms that allow each processor to generate an $O(\log n)$ bit random number at each step in the computation.

[2] The similarity assumption is the assumption that $\log n = O(\log C)$ where C is the largest edge cost. See [18] for details.

a matching in a graph with large edge costs to the problem of finding a sequence of matchings in a sequence of graphs, each of which has small edge costs. Scaling was first introduced by Edmonds and Karp [15] and has recently been a part of efficient sequential algorithms for shortest paths [3, 18], maximum flow [18, 4, 5], minimum-cost flow [15, 23, 2, 39, 19], and matching [19, 4]. In parallel computation scaling has received somewhat less attention [19, 21]. Our algorithm combines ideas involving scaling and dual variables from the sequential algorithms of Gabow [18] and Gabow and Tarjan [19], with the parallel matching algorithms of Mulmuley, Vazirani, and Vazirani [38] and Karp, Upfal, and Wigderson [30].

Chapter 2

Finding a Maximal Set of Edge Disjoint Cycles

In this chapter, we introduce a new graph decomposition technique, that of decomposing a graph into a *maximal set of edge-disjoint cycles*. Alternatively, this can be viewed as a set of cycles whose removal from a graph renders the graph acyclic. In Section 2.1, we discuss the concepts of a *cycle space* and a *cycle basis*, and show their relationship to a spanning forest of a graph. In Section 2.2, we use these concepts to give a parallel algorithm that finds a maximal set of edge-disjoint cycles in $O(\log n)$ time on $(m+n)/\log n$ processors. Finally, in Section 2.3, we use the algorithm of Section 2.2 as a subroutine in an algorithm that finds a maximal set of edge-disjoint cycles in $O(\log n \log C)$ time on a multigraph, where C is the largest edge multiplicity.

This chapter is joint work with Philip Klein.

2.1 Preliminaries

Let $G = (V, E)$ be an n-node, m-edge undirected graph with node set V and edge set $E = \{e_1, \ldots, e_m\}$. A *maximal set of edge-disjoint cycles* of $G = (V, E)$ is a set of cycles $C_1, \ldots, C_k$ s.t. $C_i \subseteq E$, $C_i \cap C_j = \emptyset$ for $i \neq j$, and the graph $G' = (V, E - \cup_i C_i)$ is acyclic. Let $\{0,1\}^E$ denote the m-dimensional vector space over $GF(2)$. Each subgraph H of G corresponds to a vector $\mu(H) = (\mu_1(H), \ldots, \mu_m(H))$, where $\mu_i(H) = 1$ if edge e_i appears in H, and $\mu_i(H) = 0$ otherwise. Further, for any edge e, let $\mu(e)$

be the vector corresponding to the subgraph containing only edge e. 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, the following fact is well-known:

Fact 2.1.1 *Let $\mathcal{C}(G) = \left\{\mu^1, \ldots, \mu^l\right\}$ be the vectors corresponding to $\{H_1, \ldots, H_l\}$, the set of all even degree subgraphs of G. Then $\mathcal{C}(G)$ is a vector subspace of $\{0,1\}^E$.*

We will call this vector subspace the *cycle space* of G and denote it by $\mathcal{C}(G)$. Further, we will use the following simple corollary of Fact 2.1.1:

Fact 2.1.2 *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.*

Note that this cycle space can contain as many as 2^m vectors, and is, in this form, impractical algorithmically.

Thus, we will focus on finding a *cycle basis* of the cycle space $\mathcal{C}(G)$, i.e. a set of linearly independent vectors that span the cycle space. Clearly, the number of vectors in any basis is equal to the dimension of the space, and hence is independent of the choice of the basis. In fact, the size of a cycle basis can be well characterized:

Lemma 2.1.3 ([9]) *Let G be a graph with p connected components. The cardinality of any cycle basis of G is exactly $m - n + p$.*

We proceed to characterize an easily found basis.

Let F be a spanning forest 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 $C(e)$ as

$$\mu(C(e)) = \mu(e) + \mu(P(v)) + \mu(P(w)), \tag{2.1}$$

because the portions of $P(v)$ and $P(w)$ from $lca(v, w)$ to the root coincide and cancel each other. We can now characterize a particular basis $\mathcal{B}$:

Lemma 2.1.4 *Let graph G have spanning forest F. Then $\mathcal{B} = \{\mu(C(e))|e \notin F\}$ is a cycle basis of G.*

Proof: Let $e_1, \ldots e_k$ be the cycle-edges of G with respect to F. Then each vector $\mu(C(e_i))$ in $\mathcal{B}$ contains exactly one cycle-edge e_i; further, each cycle-edge appears in exactly one vector in $\mathcal{B}$. Thus, this collection of vectors is linearly independent, and its cardinality is $m - (n - p)$, since any spanning forest contains $n - p$ edges. Thus, by Lemma 2.1.3, $\mathcal{B}$ is a cycle basis. ■

2.2 An Algorithm for Graphs

We will use the cycle basis $\mathcal{B}$ as a building block for a simple algorithm for finding a maximal set of edge-disjoint cycles. First, we find a rooted spanning forest, $\hat{F}$, of G. Second, we let $\mathcal{B} = \left\{\mu^1 \ldots \mu^k\right\}$ be the cycle basis defined in Lemma 2.1.4 and compute the subgraph H defined by

$$\mu(H) = \sum_{\mu^i \in \mathcal{B}} \mu^i. \tag{2.2}$$

Finally, we decompose H into a set of edge-disjoint cycles. The algorithm appears in Figure 2.1.

To show that this is indeed a maximal set of edge-disjoint cycles, we will prove the following lemma:

Lemma 2.2.1 *Let H be defined by $\mu(H) = \sum_{\mu^i \in \mathcal{B}} \mu^i$. Then H is a maximal set of edge-disjoint cycles.*

Input: Undirected graph G.
Output: H, a maximal set of edge-disjoint cycles of G.
1 Choose a rooted spanning forest $\hat{F}$ of G.
2 Determine the subgraph H of G by (2.2).
3 Decompose H into edge-disjoint cycles.

Figure 2.1: Algorithm *Maximal Cycles*

Proof: By repeated application of Fact 2.1.2, H is an even degree subgraph, and hence, can be decomposed into edge-disjoint cycles. Now we show that it is maximal. Let $e_1, \ldots e_k$ be the cycle-edges of G with respect to F. Then, by equation 2.1 and the definition of $\mathcal{B}$,

$$\mu(H) = \sum_{\mu^i \in \mathcal{B}} \mu^i = \sum_{e \notin F} \mu(C(e)) \tag{2.3}$$

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. All edges not in H must be in F, so $G - H \subseteq F$, and thus $G - H$ is acyclic. ■

Now, we focus on the time it takes to implement algorithm *Maximal Cycles*. Step 1 can be implemented in $O(\log n)$ time using $(m+n)/\log n$ processors by the spanning tree algorithm of Jung [29]. Step 3 can be implemented in $O(\log n)$ time using $(n+m)/\log n$ processors by the list-ranking algorithm of Cole and Vishkin [11] or that of Anderson and Miller [7]. A naive implementation of step 2 consists of summing at most $m - n + 1$ vectors of length m. This approach takes $O(\log n)$ time but requires about m^2 processors. We describe an alternate approach that takes advantage of the structure of the problem to achieve $O(\log n)$ time using only $(n+m)/\log n$ processors.

Recall that our goal is to achieve a characterization of which edges are in $\mu(H) = \sum_{\mu^i \in \mathcal{B}} \mu^i$. First observe that every edge of G that is not in the forest F is automatically in the subgraph H. Thus, we need only determine which edges of F are contained in H. We use equation 2.1 to rewrite equation 2.3 as

$$\mu(H) = \sum_{e=(v,w) \notin F} [\mu(e) + \mu(P(v)) + \mu(P(w))]. \tag{2.4}$$

Now focus on $\mu_i(H)$, the i^{th} component of $\mu(H)$. Because addition over $GF(2)$

Figure 2.2: To obtain F_{split}, split into two every edge not in the forest $\hat{F}$. (The shaded edges are a spanning tree rooted at r.)

is equivalent to logical exclusive-or, we can express equation 2.4 as

$$\mu_i(H) = 1 \Leftrightarrow \bigoplus_{e=(v,w)\notin F} \mu_i(e) \oplus \mu_i(P(v)) \oplus \mu_i(P(w)). \tag{2.5}$$

Let (x, y) be an edge of $\hat{F}$ corresponding to $\mu_i(H)$ with y the parent of x. For any $e = (v, w) \notin F$, $\mu_i(e) = 0$, and $\mu_i(P(v)) = 1$ only if v is a descendant of x.

Let

$$\#d(x) = \sum_{v \text{ a descendant of } x} |\{(v, w) \ : \ (v, w) \notin F\}|. \tag{2.6}$$

We can use equation 2.6 to rewrite equation 2.5 as

$$(x, y) \in H \Leftrightarrow \#d(x) \equiv 1 \bmod 2. \tag{2.7}$$

In order to compute H, therefore, it suffices to compute $\#d(x)$ for each node x of G.

We compute $\#d(x)$ by the following procedure. Let F_{split} be the rooted forest obtained from $\hat{F}$ and G by splitting in two each edge not in $\hat{F}$, as illustrated in Figure 2.2. More formally, F_{split} is obtained from $\hat{F}$ by adding two new nodes a_i and b_i and two new edges (a_i, x) and (b_i, y) for each edge $e_i = (x, y) \in G - F$. We refer to the original nodes of G as *old nodes* and the new nodes obtained by splitting edges as *new nodes*. We then use parallel tree evaluation [37, 44, 11], to compute,

Input: Undirected graph G with spanning forest $\hat{F}$.
Output: H, an even degree subgraph of G, such that $G - H$ is acyclic.

1 Let F_{split} be obtained from G by splitting in two each edge not in $\hat{F}$

2 For each node $x \in G$, use parallel tree evaluation to compute $\#n(x)$, the number of descendant new nodes of x in F_{split}.

3 Let $H = \{(v,w) : (v,w) \notin F\} \cup \{(x,y) : \#n(x) \equiv 1 \bmod 2\}$

Figure 2.3: Computing the subgraph H

for each node x, $\#n(x)$, the number of descendants of x that are new nodes. This implementation of Step 2 appears in Figure 2.3.

The following lemma justifies this procedure:

Lemma 2.2.2 *Let $\#n(x)$ be the number of descendant new nodes of x in F_{split}, and let $\#d(x)$ be defined by equation 2.6. Then $\#n(x) = \#d(x)$, where $\#d(x)$ is computed in G. Further, $\#n(x)$ can be computed in $O(\log n)$ time on $(m+n)/\log n$ processors.*

Proof: In the rooted forest F_{split}, each of the new nodes a_i and b_i is a leaf. It is easy to see that the number of new nodes connected to an old node v is $|\{(v,w) : (v,w) \notin F\}|$. Hence for each node x of G, the number of descendants of x in F_{split} that are new nodes is exactly $\#d(x)$. Thus to compute $\#d(x)$, it suffices to compute $\#n(x)$ in F_{split}. Further, observe that the number of nodes in F_{split}, including new nodes, is at most $n + 2m$. Thus, using parallel tree evaluation, we can compute $\#n(x)$ for all old nodes x of G simultaneously, in $O(\log n)$ time using $(m+n)/\log n$ processors.

Combining all the above results, we get the following theorem:

Theorem 2.2.3 *Algorithm* Maximal Cycles*, with Step 2 implemented as in Figure 2.3 finds a maximal set of edge-disjoint cycles in an undirected graph in $O(\log n)$ time on $(n+m)/\log n$ processors.*

Proof: Immediate from Lemmas 2.2.1 and 2.2.2 and the fact that computing $\#n(x)$ is the dominant step in computing subgraph H. ■

2.3 An Algorithm for Multigraphs

In order to model the notions of flow and capacity that arise in network optimization problems, we consider *multigraphs*, graphs in which there may be many edges connecting a given pair of nodes. A multigraph can be succinctly represented as $G = (V, E, m)$, where $G' = (V, E)$ is an ordinary (*simple*) graph, and $m : E \rightarrow \mathbf{Z}^{+}$ assigns a non-negative *multiplicity* to each edge in E. Thus for each edge $e = (x, y)$ of the simple graph G', there are $m(e)$ edges with the same endpoints x and y in the multigraph G. Since the multiplicity of an edge changes over the course of the algorithm, we will use $m_0(e)$ to denote the multiplicity of edge e in the input graph, and $m(e)$ to denote the current multiplicity of edge e. Let M be the maximum multiplicity of any edge in the multigraph G.

We shall give a parallel algorithm to find a maximal collection of edge-disjoint cycles in a multigraph G, where no cycle is permitted to contain more than one edge with the same endpoints. Observe that if M is very large, we might have to remove an enormous number of cycles from G in order to render G acyclic. Therefore, the output of the algorithm shall consist of a collection of pairs of the form (C, m), where C is a cycle and m is taken to be the multiplicity of the cycle. Such a collection is a solution to the maximal edge-disjoint cycles problem for G if the following two conditions are satisfied:

M.1: For every edge e of G', the sum of the multiplicities of cycles C in which e occurs is at most the original multiplicity of e in G.

M.2: The set of edges e for which the above inequality is strict form an acyclic subgraph of G'.

Our parallel algorithm to find such a solution takes $O(\log n \log M)$ time using $(m + n)/\log n$ processors.

Define $m(C)$, the *multiplicity of cycle* C, to be $\min_{e \in C} \{m(e)\}$, and define the *maximum cycle multiplicity* of a multigraph to be the maximum multiplicity of any cycle in G, i.e. $\max_{C \in \mathcal{C}} m(C)$, where $\mathcal{C}$ is the set of all cycles in G'. Our general approach will be to remove cycles of high multiplicity. Our algorithm ensures that the maximum cycle multiplicity is a non-increasing function of time. More specifi-

Input: Multigraph $G = (V, E, m_0)$.
Output: S, a maximal set of edge-disjoint cycles.

1 Let $\Delta \leftarrow 2^{\lceil \log(M+1) \rceil}$. Let $S \leftarrow \emptyset$. Let $m \leftarrow m_0$.
2 While $\Delta > 1$
3   Let G^Δ be the graph induced on the edges of G' with $m(e) \geq \Delta/2$.
4   Let F be a spanning forest of G^Δ in which appear all edges of multiplicity at least Δ.
5   Using F, find a maximal set of edge-disjoint cycles in G^Δ, ignoring multiplicities.
6   For each cycle C found in step 5,
7   Assign multiplicity $\Delta/2$ to C, and add it to S.
8   For each edge $e \in C$, let $m(e) \leftarrow m(e) - \Delta/2$.
9   Let $\Delta \leftarrow \Delta/2$.
10 Output the set S of cycles with multiplicities.

Figure 2.4: Algorithm *Maximal Capacitated Cycles*

cally, we will show that a routine similar to *Maximal Cycles*, applied to the proper graph, will decrease the maximum cycle multiplicity by a factor of 2. Assuming all initial multiplicities are integer-valued, only $\log M$ iterations are needed.

The algorithm maintains a variable Δ such that Δ is a strict upper bound on the maximum cycle multiplicity. In one iteration of the algorithm we consider only the edges of multiplicity at least $\Delta/2$. Furthermore, we force all edges of multiplicity at least Δ to be in the spanning tree. Once we have ensured that we satisfy these constraints, we find a maximal set of edge-disjoint cycles in the resulting graph, assign each of these cycles a multiplicity of $\Delta/2$, and then remove these cycles from G, adding them to S. The algorithm, *Maximal Capacitated Cycles*, appears in Figure 2.4. When an iteration terminates, S contains a collection of cycles with multiplicities. To show this collection is maximal, we first prove the following lemma:

Lemma 2.3.1 *At each iteration in algorithm* Maximal Capacitated Cycles*, step 4 succeeds, and the maximum cycle multiplicity is less than* Δ.

Proof: The proof is by induction on the number of iterations. The basis is trivial because initially Δ exceeds every edge-multiplicity. Suppose that after i iterations, the lemma holds, and the loop has not terminated. We will prove the lemma holds after the $i+1$st iteration. Since there is no cycle of multiplicity Δ, the edges of multiplicity at least Δ form an acyclic subgraph of G. Hence a spanning forest F containing every such edge can be constructed in Step 4. Consider the non-tree edges with multiplicity at least $\Delta/2$. Since they are not in the forest, these edges each have multiplicity less than Δ. In steps 5 through 8, we reduce the multiplicity of each of these edges by $\Delta/2$. Hence after step 8, every non-tree edge has multiplicity less than $\Delta/2$. Thus, the edges of multiplicity greater than or equal to $\Delta/2$ form an acyclic subgraph, and after Δ is halved in step 9, the lemma still holds. ∎

Given this lemma, we can prove the following theorem:

Theorem 2.3.2 *Algorithm* Maximal Capacitated Cycles *finds a maximal set of cycles in an n-node undirected multigraph G in $O(\log n \log M)$ time on $(m+n)/\log n$ processors.*

Proof: First we show that when the algorithm terminates, S is a maximal set of edge-disjoint cycles. It is easy to see that condition M.1 is maintained throughout the algorithm. Thus, when the algorithm terminates, for each edge e, the current value of $m(e)$ is $m_0(e) - \sum_{\{C:e\in C\}} m(C)$, where $m(C)$ is the multiplicity of cycle C in S, and $m(e) \geq 0$. Furthermore, since $\Delta \leq 1$, by Lemma 2.3.1 there are no cycles of multiplicity greater than or equal to one and the graph of edges with $m(e) > 0$ is acyclic. Thus we have satisfied condition M.2.

Now we show that we can achieve the stated resource bounds. Since Δ is initially no bigger than $2(M+1)$, and never decreases below 1, there are only $O(\log M)$ iterations. We claim that each iteration can be carried out in $O(\log n)$ time on $n+m$ processors. It is easy to see that each step, except for steps 4 and 5, takes constant time on $n+m$ processors and hence takes $O(\log n)$ time on $(n+m)/\log n$ processors. By looking at the spanning tree algorithm of [29], it is easy to see that given an acyclic set of edges that have to be in the spanning tree, the problem only becomes easier, thus steps 4 and 5 can be done in the same time as algorithm *Maximal Cycles*. ∎

We conclude with three observations. First, assuming *similarity*, i.e. $M = O(n^k)$ for some constant k [18], this algorithm runs in $O(\log^2 n)$ time on $(m+n)/\log n$ processors. Second, observe that when we removed a cycle we always assigned it a multiplicity of $\Delta/2$. However, we could in general assign it a higher multiplicity, namely that of the minimum-multiplicity edge in it. Since this could only serve to reduce the maximum cycle multiplicity at an even faster rate, the bounds above still hold. Finally, while this algorithm has been presented as an algorithm on multigraphs, it could also be viewed as an algorithm that works on graphs with weights or capacities on the edges, where the weights or capacities correspond to the edge multiplicities.

Chapter 3

Approximating the Minimum Cycle Cover

In this chapter, we develop 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 minimum cycle cover is a cover that uses the fewest edges possible. We give a series of parallel algorithms, each of which finds a smaller cover than the previous one. In Section 3.1, after giving the history of the problem, we give a simple algorithm that finds a cover of size $O(m + n^2)$. 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, Background, and a First Algorithm

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.

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 [26], finds a cover of size $O(m + n \log n)$ in $O(n^3)$ time. Subsequently, Itai, Lipton, Papadimitriou and Rodeh [25] 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 [27] that shows that every biconnected graph has a nowhere zero flow modulo 8, and results of Tarjan [43] and Shiloach [42] that find edge-disjoint branchings. They also conjecture that finding the minimum cycle cover is NP-complete. Alon and Tarsi [6] have developed an algorithm that finds a smaller cover, one of size at most $\min\left\{\frac{5}{3}m, m + \frac{7}{3}n - \frac{7}{3}\right\}$, and runs in $O(m + n^2)$ time. This result relies on a proof by Seymour [41] that every biconnected graph has a nowhere zero flow modulo 6. Alon and Tarsi also note that a certain graph called the Peterson graph [26, 25] 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 [30].

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 the fastest parallel algorithm to find any type of non-trivial cycle cover. 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.

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

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

Figure 3.1: Algorithm *Cycle Cover*

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 $O(m \log n)$. As in algorithm *Maximal Capacitated Cycles*, we will utilize our freedom to choose which edges to put in the spanning tree. 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 $\mathcal{C}$. 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 $\mathcal{C}$.

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 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 3.1. 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 splice out all nodes of degree two. (To "splice out" a node of degree two is to contract one of its incident edges.)

Lemma 3.2.1 *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_H be the spanning tree of $H = H(G, E(\mathcal{C}))$ 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:

<u>Case 1:</u> (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'.

<u>Case 2:</u> (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 biconnected with degree at most two, and 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 3.2.2 *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 3.2.1, there are $O(\log n)$ iterations and each iteration is the algorithm *Maximal Cycles* with some restrictions place on the choice of the spanning tree. ■

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 [10]. 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. 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 biconnected 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 [25], 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 each node v, the edge (v, w), where w is the node that minimizes $level(lca(v, w))$.

Input: A biconnected graph G and a rooted spanning tree $\hat{T}$.
Output: A biconnected 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 $level(lca(v, w))$.

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

Figure 3.2: **Algorithm** *Sparse Biconnected Subgraph*

This algorithm, *Sparse Biconnected Subgraph*, appears in Figure 3.2.

Lemma 3.3.1 *Given a biconnected graph G and a spanning tree T, algorithm* Sparse Biconnected Subgraph *computes a biconnected 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 one additional edge per node, $|E_B| \leq 2n - 1$. Further, since B contains a spanning tree, B is connected. Now we will argue that B is biconnected. Assume that B is not biconnected. This implies that there exists a node v' that is an articulation point of B, i.e. there is no edge from a descendant of v' to an ancestor of v' (descendant and ancestor are defined with respect to $\hat{T}$). This implies that for each descendant x of v' in G, the y that minimizes $level(lca(x, y))$ is also a descendant of v'. But if the y that minimizes $level(lca(x, y))$ is a descendant of x, then all y such that (x, y) is an edge must also be descendants of v'. This implies that v' is an articulation point of G, which contradicts the assumption that G is biconnected. Thus, B must be biconnected. ■

Combining Lemma 3.3.1 with Theorem 3.2.2 we get the main result of this section.

Theorem 3.3.2 *Combining algorithms* Cycle Cover *and* Sparse Biconnected 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 [40] that show how to compute lca and $level$ in the stated time bounds. ■

Our parallel algorithm translates into an efficient sequential algorithm.

Theorem 3.3.3 *A sequential implementation of algorithms* Cycle Cover *and* Sparse Biconnected 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 biconnected 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.

We conclude by observing the algorithm generalizes to multigraphs in the natural way, by replacing the algorithm of Section 2.2 with the algorithm of Section 2.3, as is summarized in the following theorem:

Theorem 3.3.4 *Let $G = (V, E, c)$ be a multigraph with maximum edge multiplicity C. Then, algorithm* Cycle Cover *with algorithm* Maximal Cycles *replaced by* Maximal Capacitated Cycles *finds a cycle cover of size $O(mC \log n)$ and runs in $O(\log^2 n \log C)$ time on $(m + n)/\log n$ processors.*

Proof: The running time follows from Theorems 3.2.2 and 2.3.2. The size of the cycle cover is $O(mC \log n)$ because there are at most mC edges in the graph and $O(\log n)$ iterations of *Maximal Capacitated Cycles.* ■

Note that we did not use a reduction to a sparse biconnected graph, as it is not clear, in this case, what exactly a sparse biconnected multigraph is.

Chapter 4

Finding an Assignment while Doing Less Work

In this chapter, we show how to use *scaling* to reduce the total amount of work needed to find a minimum perfect matching in a bipartite graph. Our techniques convert algorithms that use a number of processors dependent on the magnitude of the largest cost in the graph into algorithms that use a number of processors that is independent of the edge costs.

4.1 Preliminaries

Let $G = (V, E, c)$ be a graph with node set V, edge set E, and an integral *cost* $c(v, w)$ associated with each edge (v, w). The edges of a graph may be either *undirected* or *directed*. In the former case, we will denote an edge between node v and node w by (v, w), while in the latter case, we will denote an edge from node v to node w as $[v, w]$. A graph is *bipartite* if the nodes can be divided into two sets V_1 and V_2 such that $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, and all edges have one endpoint in V_1 and one endpoint in V_2.

A *matching* on a graph is a set M of edges, such that each node is incident to no more than one edge from M. A *perfect matching* is a matching in which every node is incident to exactly one matched edge. If the edges have costs, the cost of a matching is the sum of the costs of the edges in the matching. A *minimum perfect matching* (MPM) is the perfect matching with the smallest possible cost. In a bipartite graph, an MPM is also called an *assignment*. It will be convenient to associate an integer-valued dual variable $d(v)$ with each node v. This allows us to define $\overline{c}(v,w)$, the *reduced cost of edge (v,w), with respect to dual variables d* by $\overline{c}(v,w) = c(v,w) - d(v) - d(w)$. Let M be a matching. We say that a set of dual variables is *tight* if

$$(v,w) \in M \quad \Rightarrow \quad \overline{c}(v,w) \leq 0 \tag{4.1}$$

$$(v,w) \notin M \quad \Rightarrow \quad \overline{c}(v,w) \geq 0. \tag{4.2}$$

We say that a set of dual variables is *zero-tight* if

$$(v,w) \in M \quad \Rightarrow \quad \overline{c}(v,w) = 0 \tag{4.3}$$

$$(v,w) \notin M \quad \Rightarrow \quad \overline{c}(v,w) \geq 0. \tag{4.4}$$

We define the *work* done by an algorithm as the product of the number of processors used and the time spent.

The first algorithm for the assignment problem is Kuhn's *Hungarian algorithm* [33]. Implemented with Fibonacci heaps [16], this algorithm runs in $O(nm+n^2\log n)$ time, which remains the best known strongly polynomial algorithm for the assignment problem. Using ideas from this algorithm, the cardinality matching algorithm of Hopcroft and Karp [24], and scaling, Gabow and Tarjan [19] have developed an algorithm that runs in $O(\sqrt{n}m\log(nC))$ time. There are no known NC algorithms for the assignment problem; however, there are RNC algorithms under the assumption that the input is given in unary. The first RNC algorithm under this assumption was given by Karp, Upfal, and Wigderson [30]. An implementation of this algorithm by Galil and Pan [20] uses $(n+C')M(n)$ processors and $O(\log n \log^2(nC'))$ time where C' is an upper bound on the maximum cost of any matching, and $M(n)$ is the minimum number of processors needed to multiply two $n \times n$ matrices. Currently $M(n) = O(n^{2.376})$ [12], and trivially, $M(n) = \Omega(n^2)$. Subsequently, a faster algorithm was discovered by Mulmuley, Vazirani, and Vazirani [38], that finds an assignment in $O(\log^2 n)$ time using $nmCM(n)$ processors, where C is the largest edge cost in the input graph. As neither one of these algorithms does less work than the other on all graphs, we will give our improvements relative to both of these algorithms.

Input: $G = (V, E, w)$ and a perfect matching $M \subseteq E$.
Output: Zero-tight dual variables.
1 Let $G' = (V', E', c')$ be the directed graph with

- $V' = V \cup \{s\}$
- $E' = \{[s, v] \mid v \in V_1\} \cup \{[v, w] \mid (v, w) \notin M\} \cup \{[w, v] \mid (v, w) \in M\}$
- $c'[v, w] = \begin{cases} 0 & v = s \\ \overline{c}(v, w) & v \neq s \text{ and } (v, w) \notin M \\ -\overline{c}(v, w) & v \neq s \text{ and } (v, w) \in M \ . \end{cases}$

2 $\delta(v) \leftarrow$ shortest path distance from s to v in G'.

3 If $v \in V_1$, $\delta(v) \leftarrow -\delta(v)$.

4 $\epsilon(w) = \begin{cases} 0 & w \in V_1 \\ \overline{c}(v', w) - \delta(v') - \delta(w) & w \in V_2 \text{ and } (v', w) \in M. \end{cases}$

5 Return $\delta + \epsilon$.

Figure 4.1: Procedure *Compute Zero-Tight Duals* (V, E, w, M)

4.2 Computing Zero-Tight Dual Variables

In this section, we address the problem of finding zero-tight duals given a matching. This can be done by the following procedure. First we create a directed graph by directing all the unmatched edges from V_1 to V_2 and all the matched edges from V_2 to V_1. We give the matched edges negative costs and the unmatched edges positive costs. Further, we add an extra node s, with edges of cost 0 from s to every node in V_1. Now, we compute the shortest path distance from s to every node v, and let these distances be the dual variables. To convert these distances to zero-tight duals, we add, to each dual variable in V_2, the reduced cost of its associated matched edge. The details appear in Figure 4.1.

Lemma 4.2.1 *The dual variables computed by procedure* Compute Zero-Tight Duals *are zero-tight with respect to matching M in graph G'.*

Proof: First note that because M is a MPM, G' has no negative cycles, so the shortest path distances are well-defined. Now, we will show that the dual variables δ computed in Steps 1, 2, and 3 are tight with respect to $\overline{c}$. By the shortest path inequality and step 3,

$$(v,w) \notin M \Rightarrow \delta(w) \leq \overline{c}(v,w) + (-\delta(v)) \Rightarrow \overline{c}(v,w) - \delta(v) - \delta(w) \geq 0. \tag{4.5}$$

Similarly,

$$(v,w) \in M \Rightarrow -\delta(v) \leq -\overline{c}(v,w) + \delta(w) \Rightarrow \overline{c}(v,w) - \delta(v) - \delta(w) \leq 0, \tag{4.6}$$

so equations 4.1 and 4.2 are satisfied. Now consider the reduced costs of the edges with respect to dual variables $\delta + \epsilon$. Because the $\epsilon(w)$'s are either the reduced costs of matched edges or 0 they are always non-positive. Thus, if $(v,w) \notin M$, its new reduced cost is

$$\begin{array}{lll} & \overline{c}(v,w) - (\delta(v) + \epsilon(v)) - (\delta(w) + \epsilon(w)) & \\ = & \overline{c}(v,w) - \delta(v) - \delta(w) - (\epsilon(v) + \epsilon(w)) & \\ \geq & \overline{c}(v,w) - \delta(v) - \delta(w) & (\text{because } \epsilon(v) \leq 0 \ \forall i) \\ \geq & 0 & \text{by equation 4.5.} \end{array}$$

If $(v,w) \in M$, then its new reduced cost is

$$\overline{c}(v,w) - \delta(v) - (\delta(w) - (\overline{c}(v,w) - \delta(v) - \delta(w))) = 0.$$

Thus, the dual variables satisfy conditions 4.3 and 4.4, and are indeed zero-tight. ■

4.3 A Scaling Algorithm

Now we give the complete algorithm that combines scaling, the procedure *Compute Zero-Tight Duals*, and a subroutine for computing an MPM. The algorithm proceeds in $O(\log n)$ iterations. At the beginning of each iteration, one bit is added to the costs and dual variables. Then a perfect matching and zero-tight dual variables are found on the graph with edges of reduced cost no greater than $2n$. The new dual variables are added to the old ones and the iteration terminates. The key to the efficiency of this algorithm is step 4, where we ignore edges with reduced cost greater than $2n$. It remains to be shown that this does not change the value of the MPM. The details of the algorithm appear in Figure 4.2.

Input: $G = (V, E, c_0)$ an undirected bipartite graph with bipartition V_1 and V_2 and cost $c_0(v, w)$ on edge (v, w). Assume that G contains a perfect matching.
Output: A minimum perfect matching M.

1 $\quad d(v) \leftarrow \begin{cases} \frac{1}{2} & v \in V_1 \\ 0 & v \in V_2. \end{cases}$
$\quad c(v, w) \leftarrow 0 \ \ \forall (v, w) \in E$.
$\quad$ Let $C = \max_{(v,w) \in E}\{|c_0(v, w)|\}$.

2 $\quad$ For $l = 1$ to $\lceil \log_2 C \rceil$

3 $\quad\quad d(v) \leftarrow \begin{cases} 2d(v) - 1 & v \in V_1 \\ 2d(v) & v \in V_2. \end{cases}$
$\quad\quad c(v, w) \leftarrow 2c(v, w) +$ (the l^{th} signed bit of $c_0(v, w)$) $\forall (v, w) \in E$.

4 $\quad\quad$ Let $E' = \{(v, w) \,|\, (v, w) \in E$ and $\overline{c}(v, w) \leq 2n\}$.

5 $\quad\quad$ Compute M, a MPM in $G' = (V, E', \overline{c})$.

6 $\quad\quad \Delta \leftarrow$ *Compute Zero-Tight Duals* $(V, E', \overline{c}, M)$.

7 $\quad\quad d(v) \leftarrow d(v) + \Delta(v) \ \ \forall v \in V$.

8 $\quad$ Output M, a minimum perfect matching.

Figure 4.2: Algorithm *Assignment* (V, E, c_0). Letting d be non-integral is simply for ease of presentation. Note that d immediately becomes integral in step 3 and remains integral throughout the remainder of the algorithm.

Lemma 4.3.1 *The graph $G = (V, E, w)$ formed in step 4 of algorithm* Assignment *always contains a MPM of total cost no more than $2n$. Further, $\forall (v, w) \in E$, $\overline{c}(v, w) \geq 0$.*

Proof: We will prove this by induction on the number of iterations of the loop in step 2. During the first iteration, all costs are either 0 or 1, so the lemma is true. Assume that it is true after step 4 on iteration $l - 1$. Then, by Lemma 4.2.1, the costs $\overline{c}$ are zero-tight with respect to Δ. Because

$$\begin{aligned} & \overline{c}(v, w) - \Delta(v) - \Delta(w) \\ = \ & (c(v, w) - d(v) - d(w)) - \Delta(v) - \Delta(w) \\ = \ & (c(v, w) - (d(v) + \Delta(v)) - (d(w) + \Delta(w)) \end{aligned}$$

it is also true that c is zero-tight with respect to $d + \Delta$.

Thus, in the graph $\hat{G} = (V, E, c)$, the reduced cost of M with respect to $d + \Delta$ is 0 and the reduced cost of every edge is non-negative. So at the start of iteration

l, all the edges have non-negative reduced cost and the MPM has reduced cost 0. Now consider the effect of adding a new bit of cost and updating the dual variables in Step 3. Let the subscripts *old* and *new* refer to the old and new values of the variables.

$$\begin{aligned} & c_{new}(v,w) - d_{new}(v) - d_{new}(w) \\ = \; & 2c_{old}(v,w) + (\text{-1 or 0 or 1}) - (2d_{old}(v) - 1) - 2d_{old}(w) \\ = \; & 2(c_{old}(v,w) - d_{old}(v) - d_{old}(w)) + (\text{0 or 1 or 2}) \end{aligned}$$

Letting $\overline{c}(v,w)$ be the reduced cost of edge (v,w) we conclude that

$$2\overline{c}_{old}(v,w) \le \overline{c}_{new}(v,w) \le 2\overline{c}_{old}(v,w) + 2 \;\; \forall (v,w) \in E. \tag{4.7}$$

Since we have just shown that all the old reduced costs are positive, it is clear that all the new reduced costs are also positive. Now consider the new cost of the matching from the previous iteration. Using the second inequality in equation 4.7, we see that

$$\sum_{(v,w)\in M} \overline{c}_{new}(v,w) \le \sum_{(v,w)\in M} 2\overline{c}_{old}(v,w) + 2 \le \sum_{(v,w)\in M} 2 \le 2n.$$

Thus the condition is satisfied after Step 4 in iteration l, and the induction holds. ■

From this lemma we conclude that no edge of reduced cost greater than $2n$ can be in the MPM, thus justifying their exclusion from the matching subroutine. This leads to our main result.

Theorem 4.3.2 *Let algorithm A be a randomized parallel algorithm for MPM that uses $Cf(n,m)$ processors and $O(\log^k n)$ time, where $f(n,m)$ is a polynomial in n and m and k is a non-negative integer. Using algorithm* Assignment *we can convert algorithm A into an algorithm for MPM that uses $nf(n,m) + M(n)$ processors and $O((\log^k n + \log^2 n)\log C)$ time.*

Proof: First we must verify that our algorithm actually finds a MPM. From Lemma 4.3.1, we see that ignoring edges of reduced cost greater than $2n$ does not change the value of the MPM. Therefore, at each step we find a valid MPM with respect to the reduced costs. Because an MPM with respect to the reduced costs has the same value as an MPM with respect to the actual costs, in the last iteration we really are finding an MPM in the graph where the current edge costs are the same as the edge costs of the input graph, thus proving correctness. To derive the resource bounds, observe that whenever we find a MPM in step 5, $C \leq 2n$. Procedure *Compute Zero-Tight Duals* is dominated by the shortest path computation that takes $O(\log^2 n)$ time on $M(n)$ processors. All other steps in the algorithm can be implemented in constant time on $O(m+n)$ processors. Combining these observations with the fact that there are only $\log C$ iterations of the main loop, the theorem follows. ■

Corollary 4.3.3

- *Algorithm* Assignment*, combined with the matching algorithm of [30], yields a randomized parallel algorithm for computing an MPM using $n^2M(n)$ processors and $O(\log^3 n \log C)$ time.*
- *Algorithm* Assignment*, combined with the matching algorithm of [38], yields a randomized parallel algorithm for computing an MPM using $n^2mM(n)$ processors in $O(\log^2 n \log C)$ time.*

Proof: Immediate from Theorem 4.3.2 and the algorithms in [30], [20], and [38]. ■

Observe that our algorithm performs less work in the case that $C = \Omega(n^{1+\epsilon})$ for some $\epsilon > 0$. Further, our algorithm outperforms the old algorithms by a factor of $O(\frac{C}{n \log C})$, so as C gets larger, our algorithm becomes even more efficient than the previous algorithms.

We can extend this algorithm for a minimum perfect matching to one that finds a minimum-cost (not necessarily perfect) matching. Let $G = (V, E, c)$ be a graph in which we would like to find a minimum-cost matching. We employ the standard trick of using an augmented graph $G' = (V, V \times V, c')$ where $c'(v,w) = c(v,w)$ if $(v,w) \in E$ and $c'(v,w) = nC$ otherwise. It is easy to see that a minimum perfect matching in G' corresponds to a minimum-cost matching in G.

Corollary 4.3.4 *Given a graph G with maximum edge cost C, running algorithm* Assignment *on G' yields an algorithm that finds a minimum cost matching us-*

ing $n^2M(n)$ *processors and* $O(\log^3 n \log(nC))$ *time or* $n^2mM(n)$ *processors and* $O(\log^2 n \log(nC))$ *time.*

Chapter 5

Conclusions and Open Problems

We have presented a new technique for decomposing undirected graphs and have given one application: finding an approximation to the minimum cycle cover. We suspect that this technique will be useful for solving other problems as well. For example, observe that the maximal edge-disjoint cycles problem is closely related to a problem that arises in finding a minimum-cost circulation in a network, namely, finding a maximal set of weighted cycles in a positively-weighted directed graph. In this case, the weight of an edge represents the capacity of that edge, and the weight of a cycle represents the flow on that cycle. A maximal set of weighted cycles corresponds directly to a set of capacitated cycles such that, after flow is pushed around these cycles, the graph of edges that still have positive capacity is acyclic. Goldberg and Tarjan [22] solve the minimum-cost circulation problem by repeatedly finding a maximal set of weighted cycles; they show how to solve the latter problem sequentially in $O(m \log n)$ time.

In view of the application to minimum-cost circulation, it is an important open problem to determine whether there is an efficient parallel algorithm for eliminating cycles in a weighted directed graph. At present, the most efficient parallel algorithm for this problem uses a reduction to weighted non-bipartite matching, which takes $O(\log^2 n)$ time on $nmM(n)$ processors, and uses randomization[38].

We have also given an algorithm for the assignment problem that performs less work than the previously known RNC algorithms. It has the appealing feature of having the number of processors be independent of the size of the edge costs. Actual parallel machines have a fixed number of processors. Therefore, this technique gives a way to solve assignment problems with arbitrarily large edge costs without having to resort to a machine with more processors.

In contrast with previous algorithms, this algorithm only works for bipartite graphs. This is because the problem of finding tight dual variables in general graphs appears to be no easier than actually finding a matching, even sequentially [14]. However, finding dual variables is the only part of the algorithm that does not generalize to general graphs.

Bibliography

[1] A. Aggarwal and J. Park. Parallel searching in multidimensional monotone arrays. *Journal of Algorithms*, 1989. Submitted. Portions of this paper appear in *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, 1988.

[2] R. K. Ahuja, A. V. Goldberg, J. B. Orlin, and R. E. Tarjan. Finding Minimum-Cost Flows by Double Scaling. *Mathematical Programming*, 53:243–266, 1992.

[3] R. K. Ahuja, K. Melhorn, J. B. Orlin, and R.E. Tarjan. Faster algorithms for the shortest paths problem. Technical Report CS-TR-154-88, Department of Computer Science, Princeton University, 1989.

[4] R. K. Ahuja and J. B. Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37:748–759, 1989.

[5] R. K. Ahuja, J. B. Orlin, and R.E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing*, 18:939–954, 1989.

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

[7] R.J. Anderson and G.L. Miller. Deterministic parallel list ranking. In *Agaean Workshop on Computing*, pages 81–90, 1988. Published as Lecture Notes in Computer Science 319, Springer-Verlag.

[8] B. Awerbuch, 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.

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

[10] 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.

[11] R. Cole and U. Vishkin. Optimal parallel algorithms for expression tree evaluation and list ranking. In *Agaean Workshop on Computing*, pages 91–100, 1988. Published as Lecture Notes in Computer Science 319, Springer-Verlag.

[12] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progression. *J. Symbolic Computation*, 9:251–280, 1990.

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

[14] W.H. Cunningham and A.B. Marsh. A primal algorithm for optimum matching. *Mathematical Programming Study*, 8, 1978.

[15] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.

[16] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

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

[18] H. Gabow. Scaling algorithms for network problems. *Journal of Computer and System Sciences*, 31:148–168, 1985.

[19] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18:1013–1036, 1989.

[20] Z. Galil and V. Pan. Improved processor bounds for algebraic and combinatorial problems in $\mathcal{RNC}$. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pages 490–495, 1985. To appear in *Journal of the ACM*.

[21] A. V. Goldberg, S. A. Plotkin, and P. M. Vaidya. Sublinear-time parallel algorithms for matching and related problems. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 174–185, 1988.

[22] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *Journal of the ACM*, 36:873–886, 1898.

[23] A. V. Goldberg and R. E. Tarjan. Solving minimum-cost flow problems by successive approximation. *Mathematics of Operations Research*, 15(3):430–466, 1990.

[24] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal on Computing*, 2:225–231, 1973.

[25] 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.

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

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

[28] D. Johnson. Parallel algorithms for minimum cuts and maximum flows in planar networks. *Journal of the ACM*, pages 950–967, 1987.

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

[30] R. Karp, E. Upfal, and A. Wigderson. Constructing a perfect matching is in random $\mathcal{NC}$. *Combinatorica*, 6:35–48, 1986.

[31] R. M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science A*, pages 869–942. Elsevier, Amsterdam, 1990.

[32] P. Klein. *Efficient parallel algorithms for planar, chordal, and interval graphs.* PhD thesis, MIT, Cambridge, MA, August 1988.

[33] H.W. Kuhn. The hungarian method for the assignment problem. In *Naval Research Logistics Quarterly*, volume 2, pages 83–97, 1955.

[34] 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.

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

[36] 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.

[37] G. Miller and J. Reif. Parallel tree contraction and its application. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pages 478–489. IEEE, October 1985.

[38] K. Mulmuley, U.V. Vazirani, and V.V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.

[39] H. Röck. Scaling techniques for minimal cost network flows. In U. Pape, editor, *Discrete Structures and Algorithms*, pages 181–191. Carl Hansen, Münich, 1980.

[40] 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.

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

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

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

[44] 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.