A Parallel Algorithm for Eliminating Cycles in Undirected Graphs

Philip Klein*
Computer Science Department
Brown University
Providence, RI

Clifford Stein †
Laboratory for Computer Science
MIT
Cambridge, MA

Abstract

We give an efficient parallel algorithm for finding a maximal set of edge-disjoint cycles in an undirected graph. The algorithm can be generalized to handle a weighted version of the problem.

Keywords: parallel algorithms, undirected graphs, cycles, Euler tour.

1 Introduction

The first problem in graph theory was posed by Euler when he asked the now famous *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 has an Eulerian cycle if and only if the graph is connected and every vertex 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 [2] have given a parallel

---

*Research partially supported by ONR grant N00014-88-K-0243 and DARPA grant N00039-88-C0113 at Harvard University

†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 to solve this problem in $O(\log n)$ time using $n + m$ processors, for graphs with n nodes and m edges.

In this paper, we consider the closely related problem of finding a maximal set of edge-disjoint cycles in a graph. 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 on $n+m$ processors on a concurrent-read concurrent-write parallel random access machine (CRCW PRAM) [10]. In fact, using randomization, our algorithm can be implemented to run in $O(\log n)$ time using only $(n + m)/\log n$ processors. Since the problem requires $\Omega(n + m)$ operations in the worst case, the resulting randomized algorithm is optimal in its use of parallelism.

The maximal edge-disjoint cycles problem is inspired by 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 which still have positive capacity is acyclic. Goldberg and Tarjan [9] 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.

While we cannot solve this problem efficiently, we can solve the undirected version of it by generalizing our algorithm for eliminating cycles in undirected graphs to handle 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 $O(\log n \log U)$ time on $n + m$ processors of a CRCW PRAM, where U is the largest weight in the graph.

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 $Mn^{3.5}m$ processors, and uses randomization[12] .

2 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. 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 $\mathcal{B}$.

3 The Algorithm

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 every 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 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.

Step 1 can be implemented in $O(\log n)$ time using $n+m$ processors by the spanning tree algorithm of Shiloach and Vishkin [13], or using $(n+m)/\log n$ time by the randomized algorithm of Gazit [7]. 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 [4] or that of Anderson and Miller [1]. 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 approach that takes advantage of the structure of the problem to achieve $O(\log n)$ time using only $(n+m)/\log n$ processors.

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 (1) to rewrite (2) as

$$\mu(H) = \sum_{e=(v,w)\notin F} [\mu(e) + \mu(P(v)) + \mu(P(w))] \tag{3}$$

Let (x,y) be an edge of $\hat{F}$, with y the parent of x. Equation (3) implies that (x,y) is in H if and only if the mod 2 sum

$$\sum_{v\text{a descendant of } x} |\{(v,w) \ : \ (v,w) \notin F\}| \tag{4}$$

is 1. In order to compute H, therefore, it suffices to compute the sum (4) for each node x of G.

Let A be the rooted forest 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 $e_i = (x,y) \in G - F$. That is, A is obtained from G by splitting in two each edge not in $\hat{F}$, as illustrated in Figure 1. Observe that the number of nodes in A, including new nodes, is at most $n+2m$.

In the resulting rooted forest A, each of the new nodes a_i and b_i is a leaf of A. 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 A that are new nodes gives the value of (4). Using Tree Tour [14], we can compute these values for all old nodes x of G simultaneously, in $O(\log n)$ time using $(n+m)/\log n$ processors.

Thus we have proved the following theorem:

Figure 1: To obtain A, split into two every edge not in the forest $\hat{F}$. (The shaded edges are a spanning tree rooted at r.)

Theorem 3.1 We can find a maximal set of edge disjoint cycles in an undirected graph in $O(\log n)$ time on m processors deterministically, or in the same time using $m/\log n$ processors randomly.

4 An Algorithm on 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 $G = (V, E, u)$, where $G' = (V, E)$ is an ordinary (*simple*) graph, and $u : E \rightarrow \mathbf{Z}^+$ assigns a non-negative *multiplicity* to each edge in E. Thus for each edge $e = (x, y)$ of the simple graph G', there are $u(e)$ 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 $u_0(e)$ to denote the multiplicity of edge e in the input graph, and $u(e)$ to denote the current multiplicity of edge e. Let U 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 U 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, d), where C is a cycle and d is taken to be the multiplicity of the cycle. Such a collection is a solution to the maximal disjoint cycles problem for G if the following two conditions

```
Input: Multigraph $G = (V, E, u_0)$.
Output: $S$, a maximal set of edge disjoint cycles.
1   Let $\Delta = 2^{\lceil \log(U+1) \rceil}$. Let $S = \emptyset$. Let $u = u_0$.
2   While $\Delta > 1$
3     Let $G^{\Delta}$ be the graph induced on the edges of $G'$ with $u(e) \geq \Delta/2$.
4     Let $L$ be the set of edges with $u(e) \geq \Delta$.
5     Let $F$ be a spanning forest of $G^{\Delta}$ such that $F \supseteq L$.
6     Using $F$, find a maximal set of edge disjoint cycles in $G^{\Delta}$, ignoring multiplicities.
7     For each cycle $C$ found in step 6,
8       Assign multiplicity $\Delta/2$ to $C$, and add it to $S$.
9       For each edge $e \in C$, let $u(e) := u(e) - \Delta/2$.
10    Let $\Delta := \Delta/2$.
11  Output the set $S$ of cycles with multiplicities.
```

Figure 2: **Maximal Set of Capacitated Cycles Algorithm**

are satisfied:

1. For every edge e of G', the sum of the multiplicities of cycles C in which e occurs it at most the original multiplicity of e in G.
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 $n + m$ processors.

Our approach to solving the maximal disjoint cycles problem will involve the use of *scaling*, first introduced by Edmonds and Karp [5]. Define the *maximum cycle multiplicity* of a multigraph to be

$$\max_{C \in \mathcal{C}} \min_{e \in C} \{u(e)\},$$

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 specifically, we will show that a routine similar to the one of the previous section, applied to the proper graph, will decrease the maximum cycle multiplicity by a factor of 2. Assuming all initial multiplicities are integer-valued, only $\log U$ iterations are needed. The algorithm appears in Figure 2.

The algorithm maintains a variable Δ such that $\Delta - 1$ is an 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. Thus when the loop terminates, S contains a collection of cycles with multiplicities. To show this collection is maximal, we first prove the following lemma:

Lemma 4.1 At each iteration in the algorithm, step 5 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 (L) form an acyclic subgraph of G. Hence a spanning forest F containing every such edge can be constructed in Step 5. 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 6 through 9, we reduce the multiplicity of each of these edges by $\Delta/2$. Hence after step 9, 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 10, the lemma still holds. ■

Given this lemma, we can prove the following theorem:

Theorem 4.2 We can find a maximal set of cycles in an n-node undirected multigraph G in $O(\log n \log U)$ time using $n + m$ processors deterministically, or in the same time using $(n + m)/\log n$ processors randomly.

Proof: First we show that when the algorithm terminates, S is a maximal set of edge disjoint cycles. When the algorithm terminates, for each edge e, the current value of $u(e)$ is $u_0(e) - \sum_{\{C:e\in C\}} d(C)$, where $d(C)$ is the multiplicity of cycle C in S. Furthermore, since $\Delta \leq 1$, the graph of edges with $u(e) > 0$ is acyclic. Thus we have satisfied conditions 1 and 2 above.

Now we show that we can achieve the stated resource bounds. Since Δ is initially no bigger than $2(U + 1)$, and never decreases below 1, there are only $O(\log U)$ iterations. We claim that each iteration can be carried out in $O(\log n)$ time on $n + m$ processors. Except for steps 5 and 6, each step takes constant time on $n + m$ processors or $O(\log n)$ time on $(n + m)/\log n$ processors. By looking at the spanning tree algorithm of [2], it is easy to see that given an acyclic set of edges which have to be in the spanning tree, the problem only becomes easier, thus steps 5 and 6 can be done in the same time as the algorithm in the previous section. ■

We conclude with two observations. First, assuming *similarity*, i.e. $M = O(n^k)$ for some constant k [6], this algorithm runs in $O(\log^2 n)$ time on m processors. As in the simple case, the processor bound can be reduced by a factor of $O(\log n)$ using randomization. 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.

5 Concluding Remarks

In the realm of sequential computation, problems of the form "find a *maximal* set of objects of a certain structure" are typically easy to solve; the interesting problems are *maximization* problems. In the realm of parallel computation, maximality problems take on new significance, because the simple sequential algorithms typically do not parallelize. Thus the study of such problems may help to delineate the boundary between what is efficiently parallelizable, and what is not.

Acknowledgments

We are grateful to David Shmoys and Éva Tardos for helpful discussions.

References

[1] R. J. Anderson and G. L. Miller, "Deterministic Parallel List Ranking," pp. 81-90, Aegean Workshop on Computing 88, published as Lecture Notes in Computer Science 319, Springer-Verlag, 1988.

[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*, 1984, pp. 249-257.

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

[4] R. Cole and U. Vishkin, “Optimal parallel algorithms for expression tree evaluation and list ranking,” pp. 91-100, Aegean Workshop on Computing 88, published as Lecture Notes in Computer Science 319, Springer-Verlag, 1988.

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

[6] H. Gabow, “Scaling algorithms for network scaling problems.” *Journal of Computer and System Sciences* 31, 1985, pp. 148-168.

[7] H. Gazit, “An optimal randomized parallel algorithm for finding connected components in a graph,” *Proc. 27th Annual Symp. on Foundations of Computer Science* (1986), pp. 492-501

[8] A. Goldberg, S. Plotkin, and G. Shannon, “Parallel symmetry-breaking in sparse graphs,” in *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, 1987, pp. 315-324.

[9] A. Goldberg, and R. Tarjan, “Finding Minimum-Cost Circulations by Canceling Negative Cycles,” in *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, 1988, pp. 388-397.

[10] R. Karp and V. Ramachandran. “A survey of parallel algorithms for shared-memory machines,” Technical Report UCB/CSD 88/408, Computer Science Division (EECS), University of California, Berkeley, CA, March 1988.

[11] G. Miller and J. Reif, “Parallel Tree Contraction and its Applications,” in *Proceedings, 26th IEEE Annual Symposium on Foundations of Computer Science*, 1985, pp. 478-489.

[12] K. Mulmuley, U.V. Vazirani, and V.V. Vazirani, “Matching is as easy as matrix inversion,” in *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, 1987, pp. 345-354.

[13] Y. Shiloach and U. Vishkin, “An $O(\log n)$ parallel connectivity algorithm,” *Journal of Algorithms 3* (1982), pp. 57-67.

[14] R. Tarjan and U. Vishkin, “Finding biconnected components and computing tree functions in logarithmic parallel time,” in *Proceedings, 25th IEEE Annual Symposium on Foundations of Computer Science*, 1984, pp. 12-22.