A Deterministic Parallel Algorithm for Planar Graphs Isomorphism *

Hillel Gazit

Abstract

We present a deterministic parallel algorithm to determine whether two planar graphs are isomorphic. The algorithm needs $O(\log(n))$ separators that have to be computed one after the other. The running time is $T = O(\log^3(n))$ time for finding separators, and the processors count is $\frac{n^{1.5}\cdot\log(n)}{T}$ – the same complexity as a deterministic single source BFS algorithm for planar graphs [19].

We also show that every planar graph has a separator such that $\sum_{v\in sep}\sqrt{d(v)} = O(\sqrt{n})$ and give a parallel algorithm to find that separator.

1 Introduction

Although it is not known if the general Graph Isomorphism problem can be solved polynomially, there are fast sequential algorithms for some classes of graphs. Hopcroft and Tarjan gave an $O(n\cdot\log(n))$ algorithm for isomorphism in planar graphs [14] using the fact that every 3-connected planar graph has a unique planar embedding. First they isolate the 3-connected components and check them for isomorphism. Then they represent each graph as a tree, in which internal vertices correspond to connected components, separating vertices, 2-connected components and separating pairs, and leaves to 3-connected components. Thus, their solution to the 3-connected planar isomorphism problem reduces the general planar graph problem to one of tree isomorphism, which has a sequential solution in linear time [1].

Gazit and Reif [12] parallelized the Hopcroft and Tarjan approach with a randomized algorithm of time complexity $O(\log(n))$ using $n^{1.5}\cdot\sqrt{\log(n)}$ processors, assuming that families of separators for both graphs are given.

Here we achieve a better result deterministically. By examining the 3-connected components for isomorphism and then mapping each graph to a tree and comparing the trees, we find whether or not the graphs are isomorphic in $O(\log^3(n))$ time using $\frac{n^{1.5}}{\log^2(n)}$ processors. The most time-consuming step is finding the separators; a faster separator algorithm will speed up this isomorphism algorithm as well.

2 Basics

Our model is the Concurrent Read Concurrent Write (CRCW) Parallel Random Access Machine (PRAM). It is a synchronized parallel-computation model where all the processors

*Research supported in part by Airforce Contract AFOSR-87-0386, Office of Naval Research Contract N00014-87-K-0310, National Science Foundation Contract CCR-8696134, DARPA/ARO Contract DAAL03-88-K-0185, DARPA/ISTO Contract N00014-88-K-0458 and NSF-DCR-85-03251.

can read and write into a common memory. In the case of concurrent writes into the same memory location, it is assumed that one of the processors succeeds arbitrarily. It is further assumed that in one time step every processor can read, write and do basic arithmetic operations with $\log(n)$-bit numbers.

In addition to the basic definitions of graph theory, we need the following definitions and theorems:

Definition 2.1 *Connectivity: A graph G is 2-connected if and only if there is no vertex v such that we can disconnect the graph by removing v. A graph is 3-connected if and only if there is no pair of vertices u and v such that we can disconnect the graph by removing u and v [13].*

Definition 2.2 *Planar embedding: A graph G is planar if it can be drawn on the plane with no pair of crossing edges. A planar embedding of a graph is such a drawing. The embedding of a graph is generally described by the cyclic order of vertices around each vertex [16]. By following this cyclic order from one edge to the next we find the faces of the graph. We can flip a face by reversing the cyclic order at every vertex; this does not change the number of faces. A 3-connected planar graph has only two planar embeddings, one a flipping of the other [5]. Fast parallel algorithms for planar embedding are presented in [20].*

Definition 2.3 *Separators: A subset of vertices B is a separator if the remaining vertices can be partitioned into to sets A and C such that there are no edges from A to C, and $|A|, |C| \leq \frac{2 \cdot n}{3}$. The sets A, B, C form a partition of G.*

We can recursively separate A and C. Each new subgraph is at most $\frac{2}{3}$ the size of the previous subgraph. We can do this until no subgraph is larger than some constant size. We associate them in a structure such that each separated subgraph is a child of its separator; that structure is a tree of separators. Then we say that every separator S in level $i > 1$ is a child of the separator in level $i-1$ which separates S from all the other separators of level i. The root of the tree is the separator B and its two children are the separator of A and the separator of C.

Lipton and Tarjan [15] showed that every planar graph has a separator B of size $\sqrt{8 \cdot n}$. In a tree of separators for a planar graph the size of every separator is at most proportional to the square root of the number of vertices it separates. By joining large neighborhoods of faces of small diameter, Gazit and Miller [10] find an $O(\sqrt{n})$ separator in a planar graph in $O(\log^3(n))$ time using $n^{1+\epsilon}$ processors for any constant $\epsilon > 0$. Since each neighborhood has a spanning tree of small radius, trees of separators within the neighborhood can be found using Miller's algorithm [16].

Definition 2.4 *Halving sets: A subset of vertices S is a halving set of a graph G if*

1. *It is connected.*
2. *Every connected component in $G - S$ has less than $\frac{n}{2}$ vertices.*

If we have a separator algorithm of time complexity T then we can find a halving set in time $T \cdot \log(n)$. If we use Gazit and Miller's algorithm [10] then $O(\log^3(n))$ is enough because the neighborhoods we used to find the first separator serve for subsequent separators.

Our approach parallelizes the Hopcroft and Tarjan algorithm. A randomized parallelization was achieved by Gazit and Reif [12]; this algorithm is deterministic. We use the fact, demonstrated herein, that every planar graph has a separator such that $\sum_{v \in sep} \sqrt{d(v)} = O(\sqrt{n})$ and present a parallel algorithm to find it. This generalizes the Lipton and Tarjan result, and may prove significant for communication and VLSI problems.

Our algorithm requires:

1. Finding the 3-connected components of each graph.

2. Examining them for isomorphism. This is the heart of the paper.

3. Building a tree of components for each graph according to the algorithm of Hopcroft and Tarjan [13] [14].

4. Comparing the trees for isomorphism.

Parallel algorithms exist for finding 3-connected components with the CRCW model [6]. They have the same complexity as the CRCW connectivity algorithms– $O(\log(n))$ time and $\frac{n+m}{\log(n)}$ processors for randomized algorithms and $\frac{m+n}{\log(n)} \cdot \alpha(m,n)$ for the deterministic algorithm, where $\alpha(m,n)$ is the inverse Ackerman function.

3 Deterministic Sampling

To find isomorphism of the 3-connected components efficiently, we deterministically select a sample of edges that ensures the choice of some pair of corresponding edges from isomorphic graphs. Starting from vertices adjacent to these edges, which we call sources, we build BFS trees in lexicographic order as was done in the randomized algorithm [17].

Our sampling technique is based on halving sets.

Lemma 3.1 *Two halving sets of the same connected graph share at least one vertex.*

Proof: Assume for contradiction that the Lemma is false, and that we have two halving sets of G, S_1 and S_2, with no common vertex. All vertices of S_1 are in a single connected component, C_1, which must be in $G - S_2$. But S_2 is disconnected from all parts except C_1, and therefore the size of the largest connected component is at least $n - |C_1| > \frac{n}{2}$, a contradiction. □

Therefore if G_1 and G_2 are isomorphic graphs for which halving sets are known, at least one vertex in the halving set of G_1 is isomorphic to some vertex in the halving set of G_2.

We find a halving set by looking for no more than $\log_{\frac{3}{2}}(n)$ successive separators, using binary search. We find a cycle separator in the graph; if it is a halving set, we have succeeded; otherwise, we find a separator in the larger of the separated subgraphs. If the two separators are connected, we have succeeded, otherwise there is a halving set between them. We find a separator in that set and proceed as above until we succeed.

Theorem 3.2 *If finding a cycle separator of $O(\sqrt{n})$ takes $T \geq \log(n)$ time using P processors then we can find a halving set in $T \cdot \log(n)$ time using $\frac{P}{\log(n)}$ processors. The size of the halving set is $O(\log(n))$.*

Proof: Each separator algorithm takes at least $O(n)$ (input size) operations. The size of the part we separate decreases geometrically, and therefore we find a halving set within $\log_{\frac{3}{2}}(n)$ iterations. By reassigning processors after each iteration we can increase the speed by a factor of $\frac{3}{2}$ at each iteration.

The size of the halving set is bounded by the sum of the sizes of all the separators that are found, yielding a trivial upper bound of $\sum_{i=0}^{\infty} O\left(\sqrt{\left(\frac{2}{3}\right)^i \cdot n}\right) = O(\sqrt{n})$. □

Lemma 3.3 *We can find a halving set in $O(\log^3(n))$ time using $n^{1+\epsilon}$ processors.*

Proof: This is a direct corollary of Gazit and Miller's algorithm [10], which finds neighborhoods and a maximal independent set of the neighborhoods. We avoid the problem of face size, on which it depends, by triangulating the graph. □

We know we need check only one edge from each vertex of the halving set of G_1 against all edges common to the halving set vertices of G_2. In a 3-connected graph we can

do better by making using the uniqueness (up to flipping) of the embedding. After embedding, we consider a block of $\lceil\sqrt{d(v)}\rceil$ edges from the first vertex and a set of edges spaced $\lfloor\sqrt{d(v)}\rfloor$ apart around the second vertex. If the two vertices are isomorphic at least one edge in the first sample must correspond to an edge in the second sample.

Lemma 3.4 *Given halving sets for G_1 and G_2,*
we can find an isomorphic edge deterministically, by picking $\sum_{v\in halv\ set\ for\ G_1}\sqrt{d(v)}+\sum_{v\in halv\ set\ for\ G_2}\sqrt{d(v)}$ edges.

Proof: The proof follows from the above discussion. □

Corollary 3.5 *If every vertex in the two halving sets has either a constant degree or an $O(n)$ degree, it suffices to check $O(\sqrt{n})$ edges.*

proof: Only a constant number of vertices can have degree $O(n)$. □

Prioritizing vertices of low degree is even more efficient. We combine Miller's [16] algorithm with that of Lipton and Tarjan [15] to favor vertices of low degree:

Assign to each edge (u,v) of G the length $\sqrt{d(v)}+\sqrt{d(u)}$ and construct the following planar graph G':

1. In the middle of every edge (u,v) add a vertex uv.
2. The new edge (v,uv) (which we call a "half-edge") has length $\sqrt{d(v)}$.
3. For every vertex v, add the loop half-edge (v,v).

Note that this preserves the distance between vertices from G.

Now solve the single-source all shortest-paths problem in G' from some arbitrary source s. It suffices to find a solution in time $T \geq O(\log(n))$ by the algorithm of Pan and Reif, and therefore $\frac{n^{1.5}\cdot\log(n)}{T}$ processors are enough.

From this solution, for each original vertex v we derive $val(v)$ as the distance from s to v.

Also, for each original vertex v, we find the distance $\min(v)$ from s to the nearest half-neighbor uv of v, and the distance $\max(v)$ from the source to the most distant half-neighbor uv of v.

Then,

Lemma 3.6
$\sqrt{d(v)} \leq max(v) - min(v) \leq 2\cdot\sqrt{d(v)}$.

Proof: The upper bound follows immediately from the fact that the distance between any two half-neighbors of v is at most $2\cdot\sqrt{d(v)}$. To find the lower bound, recall that if v is not the source, then one of v's neighbors is its parent in a shortest path tree. If v is the source then the value of any neighbor of v is $\sqrt{d(v)}$.

This allows us to generalize Lipton and Tarjan's idea of BFS layers as a base for a separator [15]. Following them, we layer the graph:

Every vertex v belongs to every layer between $\min(v)$ and $\max(v)$.

All vertices such that $\max(v) < k$ belong to layers "lower" than k. All vertices such that $\min(v) > k$ belong to layers "higher" than k. Each k defines an equivalence relation for all vertices of the graph.

Lemma 3.7 *There is no edge in G between a vertex v in a layer $< k$ and a vertex u in a layer $> k$.*

Proof: Assume that there is such an edge (u,v). By the definition of max, $val(uv)$ must be less than k, but by definition of min, $val(uv)$ must be greater than k, a contradiction. □

Selecting a layer L divides the graph into three parts:

1. vertices in lower levels
2. vertices in L
3. vertices in higher levels

The lower levels make a single connected component. L itself may be one or more connected components, as may the higher levels.

Lemma 3.8 *If the graph is triangulated then every connected component of a higher level C is separated from s by a single connected component of L.*

Proof: Assume that some vertex in C has two paths, in C, to two different connected components of L, A and B. Then there is a face in the graph with at least four vertices: the vertex in C, the vertex in A, the vertex in B, and some vertex from a lower level, contradicting the assumption of triangulation. □

This Lemma implies that the layers induce a tree of which each connected component of a layer is a vertex. This structure lets us compute the number of vertices above and below each connected component of every layer using a simple Tree-Tour [22].

We can compute the number of vertices above every connected component of every layer in $O(\log(n))$ time using optimally many processors by applying an optimal connected components algorithm [8]. Then we weight the layers so we can disconnect the graph by choosing small layers. The weight should indicate how much work will be needed if a given vertex is in the separator, i.e. $\sqrt{d(v)}$.

Lemma 3.9 *The sum of all the weights is bounded by $12 \cdot n$.*

Proof: By Lemma 3.6 every vertex can be in at most $2 \cdot \sqrt{d(v)}$ layers; the total weight is $\sum_{v \in V} \left(2 \cdot \sqrt{d(v)}\right) \cdot \sqrt{d(v)} = 4 \cdot |E| < 12 \cdot n$, since the planarity of the graph guarantees $|E| < 3 \cdot n$.

A small layer has weight $\leq 4 \cdot \sqrt{n}$; a large layer has weight $> 4 \cdot \sqrt{n}$.

Corollary 3.10 *There are at most $3 \cdot \sqrt{n}$ large layers in the graph.*

We choose two close small layers. The "top" is the first small layer such that the number of vertices above every connected component of the layer is less than $\frac{n}{2}$; the "bottom" is the closest small layer such that the weight above one of its connected components is at least $\frac{n}{2}$. There is exactly one such connected component. We cut the graph by this component and look for the parts of the top layer separated by this cut from the root; these are the vertices of the top cut.

Lemma 3.11 *Every vertex in the top cut is at distance $3 \cdot \sqrt{n}$ or less from the bottom cut.*

Proof: The proof is similar to that in [15]. The key points are that each vertex v is in at least $\sqrt{d(v)}$ layers, and by the corollary there are at most $3 \cdot \sqrt{n}$ layers between the bottom cut and the top cut. □

This is important because the length of the path represents the number of edges we must sample if the path is in the halving set. We need to find two $\frac{1}{3}$ or better separators in the domain between the bottom cut and the top cut, but these are easy exercises in finding lowest common ancestor [21], and induced cycles [16] [15].

It follows from the above discussion that:

Theorem 3.12 *We can find a halving set S in a planar graph such that $\sum_{v \in S} \sqrt{d(v)} = O(\sqrt{n})$. If we have a family of separators then the complexity of the algorithm is $O(\log(n))$ time using $n^{1.5}$ processors.*

Proof: We can solve a single source all shortest paths problem in $O(\log(n))$ time using $n^{1.5}$ processors by Pan and Reif's algorithm [19]. After we have the shortest paths we can find the layers, the connected components of each layer, and the difference, in number of vertices, between every two consecutive layers. Afterward we can compute the number of vertices above every connected component of every layer in $O(\log(n))$ time by a simple Tree Tour technique [22].

Finding the "top" and the "bottom" layers is quite simple after the previous computations were done. □

Corollary 3.13 *Every planar graph has a separator such that* $\sum_{v \in sep} \sqrt{d(v)} = O(\sqrt{n})$. *This separator can be found, sequentially, in linear time and linear space.*

Proof: The proof is similar to the proof of Theorem 3.12. The key observation is that the diameter of G' is bounded by $\sum_{v \in V} \sqrt{d(v)} \leq \sum_{v \in V} d(v) = 2 \cdot m$; therefore we can use $2 \cdot m$ buckets instead if a heap in Dijkstra's algorithm [1]. □

We now have a sampling technique that guarantees selection of corresponding edges from isomorphic graphs. We compute a BFS from each selected vertex.

4 Isomorphism of 3-connected Components

Lemma 4.1 *Given a tree of separators, we can compute BFS from k arbitrarily selected sources in* $O(\log(n))$ *time using* $O(k \cdot n + n^{1.5})$ *processors.*

Proof: We follow Pan and Reif [18] [19], dividing the graph using a tree of separators and then computing the shortest distances at each level. Details are provided in the full paper. □

Each BFS lets us construct a BFS tree as described in [12]. We order children of each vertex uniquely, from the planar embedding. Parallel algorithms for planar embedding are presented in [20]. Details of the multi-source BFS and construction of the BFS tree are given in the complete paper.

We then use the Tree-Tour algorithm of Tarjan and Vishkin to label the trees. Then we assign each labeling a number by encoding every edge as its adjacent vertices and sorting by Cole's algorithm; see [12].

This yields the following isomorphism algorithm for 3-connected components:

1. Find the embedding of each graph.
2. Triangulate the graph by adding a new vertex in the middle of every face and connecting every vertex in the face to it.
3. Build a tree of separators for each graph.
4. Build G_1' and G_2' (with half-edges) and find all shortest paths from some single source (arbitrarily chosen).
5. Find a halving set in G_1 and in G_2.
6. Flip the embedding of one graph.
7. For every vertex in the halving set of G_1, select from the list of embedded edges, $1, \ldots \lceil \sqrt{d(v)} \rceil$.
8. For every vertex in the halving set of G_2, select from the list of embedded edges $\lfloor \sqrt{d(v)} \rfloor, 2 \cdot \lfloor \sqrt{d(v)} \rfloor, \ldots$
9. For every edge in the sample, choose a direction with respect to the separator vertex.
10. Set the length of every old edge to 1 and every new edge to n.
11. In all three graphs (both originals and the flipped embedding) perform BFS using as sources all vertices pointed to by edges from the sample.
12. Compute all the BFS trees from the sampled edges and, for every tree, find a labeling of the original vertices.
13. Give to every BFS tree a label which is the sorted, with respect to the BFS numbering, list of the edges of the graph.
14. Sort the labels using Cole's algorithm [3].
15. If two labels are the same, the graphs are isomorphic.

Lemma 4.2 *The isomorphism algorithm for 3-connected planar graphs requires time* $T = O(\log^3(n))$ *and processor count* $P = \frac{n^{1.5}}{\log^2(n)}$ *processors.*

Proof: The proof follows immediately from Lemma 4.1 and Theorem 3.12. □

5 Generalization

We generalize to directed graphs simply by computing BFS trees for the underlying graph and sorting the edges as ordered pairs; all Lemmas still hold.

To generalize to all planar graphs we build a tree like that of Hopcroft and Tarjan [14] where the leaves are 3-connected components and the internal nodes are connected and 2-connected components and separating vertices and pairs, as described in [13] and [14]. We can find 2- and 3- connected components in $O(\log(n))$ time with the same (sublinear) processor complexity as the parallel connectivity algorithms [4] [6] [7] [22].

The entire algorithm is this:

1. Build a Hopcroft-Tarjan tree for each graph.
2. Classify the leaves by their numbers of edges and vertices.
3. Apply the 3-connected components isomorphism algorithm.
4. Give isomorphic 3-connected components the same label.
5. Check for tree isomorphism by deterministic tree contraction.

The planarity of the graphs and the number of edges in a tree ensure that the total number of edges in all components in bounded by $4 \cdot n$.

Lemma 5.1 *Computing BFS from k sources in each 3-connected component has at most the time and processor complexity as that of computing BFS from k sources in the original graph.*

Proof: The proof follows immediately from Lemma 4.1.

Theorem 5.2 *If we have trees of separators for two planar graphs then we can determine whether they are isomorphic in* $O(\log(n))$ *time using* $n^{1.5}$ *processors plus the complexity of tree isomorphism.*

Proof: Provided in the paper, from discussion above. □

Note that the family of separators is found in $O(\log^3(n))$ time on $n^{1+\epsilon}$ processors [9] because we keep reusing the same partition. Therefore we find the halving set in time $O(\log^3(n))$.

6 Tree Isomorphism

The problem of tree isomorphism has a simple and optimal sequential solution which is based of radix sort [1]. There are good parallel randomized solutions which are based on tree contraction; our deterministic solution is based on tree contraction as well.

The solution is based on the deterministic tree contraction algorithm of Gazit, Miller and Teng [11]. The advantage of this algorithm is that it always performs the same contraction on a given tree, even if the order of children is changed.

We will discuss several cases to explain our idea. The simplest one is a binary tree - every vertex has at most two children. In every stage of the tree contraction algorithm we first remove all leaves, and then isolate every chain and compress it. A proof that $O(\log(n))$ stages are enough was presented in [11]. The problem is how to number vertices with similar subtrees with the same number. In the

case of rake the solution is simple; we assume, inductively, that all the current leaves that were roots of isomorphic trees have the same id number. If we had n^2 space and a vertex v had children i,j then it will write its number in $M[i,j]$. Because of concurrent write we don't know which processor will win, but every processor can read $M[i,j]$ in the next iteration and set its number to that value. To reduce space to $n^{1+\epsilon}$ we use the deterministic hashing method that was used by Cole and Vishkin in their connectivity algorithm [4].

The more complicated problem is compressing the chains. The solution is to rank them first, and then to divide, deterministically to pairs, compress them by a similar method to the compression of the leaves, and continuing the process. The complexity will be log of the number of the vertices in the chain, which is the same as compressing the chain in [11]; therefore $\log(n)$ iterations will be enough.

The problem is more complicated if the degree of every vertex is not bounded. Naively waiting till all the leaves were raked will not work because the last rake will take too long. Compressing the leaves before we raked all of them has the problem that we do not know in what order to rake them.

The solution is to sort the raked leaves after every iteration, and then to compress them by the sorted order. The result is similar to deleting a constant fraction of the leaves in every iteration, and therefore $O(\log(n))$ iterations are enough.

Lemma 6.1 *If in every iteration we delete a constant fraction of the leaves and the vertices with one child then $O(\log(n))$ iterations are enough.*

Proof: The proof follows from the fact that the number of leaves is larger than the number of vertices with two children or more. See [17] for details. □

Note that we cannot actually delete about half of the number of leaves, but we still can see that the time of the algorithm is $O(\log(n))$ by the following observation: In every iteration we copy the list and delete the non-leaves by list ranking, and then we sort them. We also copy the list and delete the leaves by list ranking. In later iterations we use the last list that its ranking was completed instead of the original list. If the last leaves to be deleted take time t then we had a list of size $2^{O(t)}$. If the algorithm had run on a computer of half the speed, then the deletion of these leaves will take the same time.

Lemma 6.2 *We can check tree isomorphism, using the CRCW model, in $O(\log(n))$ time using $n \cdot \log^3(n)$ processors and $n^{1+\epsilon}$ space.*

Proof: The proof follows from the above discussion. □

We would like to reduce the processor count to an optimal number. We use a similar idea to Gazit, Miller and Teng [11] to reduce the problem size by a factor of $\log(n)$ in $O(\log(n))$ time.

The idea is as following:

1. Find, by Tree-Tour [22], the weight of every subtree.
2. Make a list of all the critical vertices v such that the weight v is less or equal to $\log(n)$, but the weight of $parent(v)$ is more than $\log(n)$.
3. Distribute the subtrees which are rooted in critical vertices between the processors so no processor will have trees with of total weight more than $2 \cdot \log(n)$.
4. Compute isomorphism between all these subtrees.
5. Do one rake, with respect to isomorphism.
6. Compress all chains, with respect to isomorphism.

Lemma 6.3 *The complexity of the algorithm is $O(\log(n))$ time using $\frac{n}{\log(n)}$ processors.*

Proof:

1. Tree-tour takes $O(\log(n))$ time.
2. Checking if a vertex is critical can be done in $O(1)$.
3. The distribution of the trees can be done by a prefix sum algorithm.
4. Computing isomorphism in the subtrees is easy because only one processor works on any tree. All the communication which is needed is a matrix to agree about a "name" for a vertex after its children were raked. Note that if a vertex is critical then none of its descendants is critical.
5. A rake can be done in $O(\log(n))$ time.
6. For compress, we first list-rank all the chains by optimal list ranking algorithm [4] [2].

Lemma 6.4 *1. After the last rake the number of leaves is at most $\frac{n}{\log(n)}$.*

2. After the last rake the number of internal vertices with 2 children or more is at most $\frac{n}{\log(n)}$.

3. After the algorithm the number of vertices with one child is at most $\frac{n}{\log(n)}$.

Proof:

1. After the last rake all the leaves had weight $\log(n)$ or more. Their number is bounded by the total weight over $\log(n)$.
2. The number of internal vertices with 2 children or more is smaller than the number of leaves.
3. After the compress every vertex with a single child has a child which should be an internal vertex with two children or more.

□

Theorem 6.5 *We can check tree isomorphism, using the CRCW model, in $O(\log(n))$ time using $\frac{n}{\log}$ processors and $n^{1+\epsilon}$ space.*

Proof: The proof follows from Lemmas 6.2, 6.3 and 6.4 □

7 Open problems

1. Developing a deterministic isomorphism *NC* algorithm with less than $n^{1.5} \cdot \log(n)$ operations.
2. Developing an optimal space, $O(\log(n))$ time tree isomorphism algorithm.

Acknowledgment: I want to thank to Rao Kosaraju and John Reif for their comments about the tree isomorphism algorithm.

I also want to thank to Mara Chibnik for editing this paper for me.

References

[1] A. Aho, J.E. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] R. Anderson and G. L. Miller. Deterministic parallel list ranking. In *AWOC 88*, pages 81–90. Aegean Workshop on Computing, 1988.

[3] R. Cole. Parallel merge sort. In *27th Annual Symposium on Foundations of Computer Science*, pages 511–516. IEEE, Oct 1986.

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

[5] S. Even. *Graph Algorithms.* Computer Science Press, 1979.

[6] D. Fussel, V. Ramachandran, and R. Thurimella. Finding triconnected components by local replacements. In *Proc. 16th ICALP, Lect. Notes in Comp. Sci. No.*, volume 372, pages 379–393. Springer-Verlag, July 1989.

[7] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph (extended version). *Accepted by SIAM J. of Computing*, 1989.

[8] H. Gazit. Optimal EREW parallel algorithms for connectivity, ear decomposition and st-numbering of planar graphs. In *Fifth International Parallel Processing Symposium*, pages 84–91, April 1991.

[9] H. Gazit and G. L. Miller. Planar separators and the Euclidean norm. In *SIGAL International Symposium on Algorithms*, Tokyo, August 1990. Information Processing Society of Japan, Springer-Verlag.

[10] H. Gazit and G. L. Miller. A deterministic parallel algorithm for finding a separator in planar graph. Technical Report CMU-CS-91-103, CMU, 1991.

[11] H. Gazit, G. L. Miller, and S. H. Teng. Optimal tree contraction in the EREW model. *Concurrent Computing*, 1988. Editors: S. K. Tawsburg, B. W. Dickinson and S. C. Schwartz.

[12] H. Gazit and J. H. Reif. A randomized parallel algorithm for planar graph isomorphism. In *2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 210–219. ACM, July 1990.

[13] J.E. Hopcroft and R.E. Tarjan. Dividing a graph into triconnected components. *Siam J. of Computing*, 2:135–158, 1973.

[14] J.E. Hopcroft and R.E. Tarjan. A $v \log(v)$ algorithm for isomorphism of triconnected planar graphs. *J. of Computer and Systems Sciences*, 7:3:323–331, 1973.

[15] R.J. Lipton and R.E. Tarjan. A separator theorem for planar graphs. *SIAM J. of Appl. Math.*, 36:177–189, April 1979.

[16] G.L. Miller. Finding small simple cycle separator for 2-connected planar graphs. *J. of Computer and System Sciences*, 32(3):265–279, June 1986.

[17] G.L. Miller and J.H. Reif. Parallel tree contraction and its applications. In *26th Symposium on Foundations of Computer Science*, pages 478–489, Portland, Oregon, 1985. IEEE.

[18] V. Pan and J. H. Reif. Extension of parallel nested dissection algorithm to the path algebra problems. In *Proc. Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, 1986.

[19] V. Pan and J. H. Reif. Fast and efficient solution of path algebra problems. *J. of Computer and System Sciences*, 38:494–510, 1989.

[20] Vijya Ramachandran and John H. Reif. An optimal parallel algorithm for graph planarity. In *30th Annual Symposium on Foundations of Computer Science*, Durham, October 1989. IEEE.

[21] B. Schieber and Uzi Vishkin. On finding the lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, December 1988.

[22] R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14(4):862–874, 1985.