ISOMORPHISM OF PLANAR GRAPHS (WORKING PAPER)

J.E. Hopcroft and R.E. Tarjan

Cornell University

Ithaca, New York

ABSTRACT

An algorithm is presented for determining whether or not two planar graphs are isomorphic. The algorithm requires $O(V \log V)$ time, if V is the number of vertices in each graph.

INTRODUCTION

The isomorphism of planar graphs is an important special case of the graph isomorphism problem. It arises in the enumeration of various types of planar graphs and in several engineering disciplines. It is important to consider the planar graph case separately since the more general problem of graph isomorphism is at present intractable. Although good heuristics exist for the general isomorphism problem, all known algorithms have a worst case asymptotic growth rate which is exponential in the number of vertices. In this paper we exhibit an efficient algorithm for testing two planar graphs for isomorphism. The asymptotic running time of the algorithm is bounded by $O(V \log V)$ where V is the number of vertices in the graphs. As a by-product we document a linear tree isomorphism algorithm. Several authors (Edmonds, Scoins, Weinburg and others), have given similar algorithms but no one has published the non-trivial details of implementing the required sorting implied by these algorithms.

Early work on the isomorphism of planar graphs is due primarily to Weinburg who developed efficient algorithms for

R. E. Miller et al. (eds.), *Complexity of Computer Computations*

isomorphism of triply connected planar graphs in time V^2 and for isomorphism of "series parallel" graphs and for trees. His algorithms can clearly be combined to give a polynomial bounded algorithm for the general problem. An improved algorithm for the isomorphism of triply connected planar graphs is given in Hopcroft (1971A) and a V^2 algorithm for isomorphism of planar graphs is given in Hopcroft and Tarjan (1971A).

The paper is divided into five sections. The first section consists of the introduction and certain graph theory terminology. The second section describes an algorithm for partitioning a graph into its unique 3-connected components in time proportional to the number of edges. The third section documents a linear tree isomorphism algorithm. The fourth section describes an algorithm for isomorphism of triply connected planar graphs in time proportional to $V \log V$. The fifth section combines the above algorithms into an isomorphism algorithm for arbitrary planar graphs. The worst case running time of the algorithm grows as $V \log V$.

The remainder of this section is devoted to terminology and notation. We assume that the reader is familiar with the more or less standard definitions of graph theory [Harary (1969A)]. A graph G consists of a finite set of vertices $\mathcal{V}$ and a finite set of edges $\mathcal{E}$. If the edges are unordered pairs of vertices then the graph is *undirected*. If the edges are ordered pairs of vertices, then the graph is *directed*. If (v,w) is a directed edge, then v is called the *tail* and w is called the *head*. A *path*, denoted by $v \overset{*}{\Rightarrow} w$, is a sequence of vertices and edges leading from v to w. A path is *simple* if all its vertices are distinct. A *cycle* is a closed path all of whose edges are distinct and such that only one vertex appears twice.

A *tree* is a connected graph with no cycles. A *rooted tree* is a directed graph satisfying the following three conditions: (1) There is exactly one vertex, called the *root*, which no edge enters. (2) For each vertex in the tree there exists a sequence of directed edges from the root to the vertex. (3) Exactly one edge enters each vertex except the root. A directed edge (v,w) in a tree is denoted $v \rightarrow w$. A path from v to w is denoted by $v \overset{*}{\rightarrow} w$. If $v \rightarrow w$, v is the *father* of w and w is a *son* of v. If $v \overset{*}{\rightarrow} w$, v is an *ancestor* of w and w is a *descendant* of v. Every vertex is an ancestor and a descendant of itself. If v is a vertex in a tree T, then T_v is the subtree of T having as its vertices all the descendants of v in T. Let G be a directed graph. A tree T is a *spanning tree* of G if T is a subgraph of G and T contains all vertices of G.

A graph G is biconnected if for each triple of distinct vertices v,w and a in $\mathcal{V}$ there is a path $p: v \overset{*}{=}> w$ such that a is not on the path p. If there is a distinct triple v,w,a such that a is on every path $p: v \overset{*}{=}> w$, then a is called an articulation point of G. Let the edges of G be partitioned so that two edges are in the same block of the partition if and only if they belong to a common cycle. Let $G_i = (\mathcal{V}_i, \mathcal{E}_i)$ where $\mathcal{E}_i$ is the set of edges in the ith block of the partition and $\mathcal{V}_i = \{v \mid \exists w \ni (v,w) \in \mathcal{E}_i\}$. Then

(i) Each G_i is biconnected.

(ii) No G_i is a proper subgraph of a biconnected subgraph of G.

(iii) Each vertex of G which is not an articulation point of G occurs exactly once among the $\mathcal{V}_i$ and each articulation point occurs at least twice.

(iv) For each $i,j, i \neq j, \mathcal{V}_i \cap \mathcal{V}_j$ contains at most one vertex; furthermore, this vertex (if any) is an articulation point.

The subgraphs G_i of G are called the biconnected components of G.

A graph is triply connected if for each quadruple of distinct vertices v,w,a,b in $\mathcal{V}$, there is a path $p: v \overset{*}{=}> w$ such that neither a nor b is on path p. If there is a quadruple of distinct vertices v,w,a,b in $\mathcal{V}$ such that there is a path $p: v \overset{*}{=}> w$ and every such path contains either a or b, then a and b are a biarticulation point pair in G.

An n-gon is a connected graph consisting of a cycle with n edges. An n-bond is a pair of vertices connected by n edges. Strictly speaking an n-bond is not a graph. We introduce it since we intend to find biarticulation point pairs and thereby divide the graph into its triply connected components. When a component is removed, it is replaced by an edge and this process can introduce multiple edges.

By a suitable modification of the above definition (see for instance Tutte (1966A)) or by breaking off only biconnected components, one can insure that the triply connected components are unique.

In deriving time bounds on algorithms we assume a random access model. In order to avoid considering specific details of the model we adopt the following notation. If $\vec{n}$ is a vector and there exist constants k_1, k_2 such that $|t(\vec{n})| \leq k_1 |f(\vec{n})| + k_2$, then we write "$t(\vec{n})$ is $O(f(\vec{n}))$".

We make use of several known algorithms. One such algorithm is called a radix sort. We can sort n integers $x_1, x_2, \ldots, x_n$ where each x_i has a value between 1 and n in time $O(n)$ by initializing n buckets. Each x_i is then placed in bucket x_i. Finally the contents of the buckets are removed in order starting with bucket 1.

A graph is stored in the computer using an adjacency structure which consists of a set of adjacency lists, one list for each vertex. The adjacency list for vertex v contains each w such that (v,w) is in $\mathcal{E}$. If G is undirected each edge is represented twice in the adjacency structure. If G is directed, then each edge (v,w) appears only once. The adjacency structure for a graph is not unique and there are as many structures as there are orderings of edges at the vertices.

Representing a graph by its adjacency structure expedites searching the graph. We make use of a particular type of search called a depth-first search. A depth-first search explores a graph by always selecting an edge emanating from the vertex most recently reached which still has unexplored edges.

Let G be an undirected graph. A search of G imposes a direction on each edge of G given by the direction in which the edge is traversed when the search is performed. Thus G is converted into a directed graph G'. The set of edges which lead to a new vertex when traversed during the search defines a spanning tree of G'. In general, the arcs of G' which are not part of the spanning tree interconnect the paths in the tree. However, if the search is depth-first, each edge (v,w) not in the spanning tree connects vertex v with one of its ancestors w. In this case G' is called a palm tree and the arcs of G' not in the spanning tree are called the fronds of G'. An edge (v,w) which is a frond is denoted by $v \dashrightarrow w$. Depth-first search can be implemented in $O(V,E)$ time, using an adjacency structure to give the next edge to be explored from a given vertex.

DETERMINING TRICONNECTIVITY

Let $G = (\mathcal{V},\mathcal{E})$ be a graph with $|\mathcal{V}| = V$ vertices and $|\mathcal{E}| = E$ edges. This section describes an algorithm for determining in $O(V,E)$ time whether G is triconnected. (Older algorithms, such as [Ariyoshi, Shirakana and Hiroshi (1971A)], require $O(V^4)$ time.) The algorithm may be extended to divide a graph into its triconnected components in $O(V,E)$ time, using Tutte's definition [Tutte (1966A)] or some other definition of triconnected components. (Tutte's definition has the advantage that it gives unique components; unique components are necessary to solve the planar isomorphism problem.)

We may assume that $|V| \geq 4$ and that G has no vertices of degree two; if $|V| \leq 4$ or G has a degree two vertex the triconnectivity problem has an immediate answer. Further, we may assume that G is biconnected; [Hopcroft and Tarjan (1971B)] describe a method for dividing a graph into its biconnected components in $O(V,E)$ time.

The triconnectivity algorithm consists of three depth-first searches. The first search constructs a palm tree P for G and calculates information about the fronds of P. An adjacency structure A is constructed for P using the information generated by the first search. The second search uses A to select edges to be explored, and calculates necessary information about P. The third search determines whether a biarticulation point pair exists.

Suppose G is searched in a depth-first fashion and that the vertices of G are numbered in the order they are reached during the search. Let vertices be identified by their numbers. Let P be the palm tree generated by the search. If $v \in \mathcal{V}$, let $LOWPT1(v) = \min(\{v\} \cup \{w \mid v \overset{*}{\rightarrow} \dashrightarrow w\})$. Let

$$LOWPT2(v) = \min[\{v\} \cup (\{w \mid v \overset{*}{\rightarrow} \dashrightarrow w\} - \{LOWPT1(v)\})] .$$

That is, $LOWPT1(v)$ is the lowest vertex reachable from v by traversing zero or more tree arcs in P followed by at most one frond. $LOWPT2(v)$ is the second lowest vertex reachable in this way. We have $LOWPT1(v) < LOWPT2(v)$ unless $LOWPT1(v) = LOWPT2(v) = v$. The numbers and LOWPT values of all vertices may easily be calculated during the first search of G.

If $v \rightarrow w$ in P, let $\phi((v,w)) = LOWPT1(w)$. If $v \dashrightarrow w$ in P, let $\phi((v,w)) = w$. Let A be an adjacency structure for P such that the adjacency lists in A are ordered according to ϕ. (Each entry in an adjacency list corresponds to an edge of P; the entries must be in order according to the ϕ values of their corresponding edges.) Such an adjacency structure A can be constructed using a single radix sort of the edges of P. See [Tarjan (1972B)]. Furthermore, A depends only on the order of the LOWPT1 values and not on the exact numbering scheme. That is, if the vertices of P are numbered from 1 to V in any manner such that $v \to w$ implies NUMBER(v) < NUMBER(w), and LOWPT1 values are calculated using the new numbers, then the possible adjacency structures A which satisfy the new LOWPT1 ordering are the same as those which satisfy the old ordering. This fact is easy to prove; see [Tarjan (1972B)].

The second search explores the edges in the order given by A, using the same starting vertex as the first search. Vertices are numbered from V to 1 as they are last examined during the search. For this numbering scheme, $v \to w$ implies NUMBER(v) < NUMBER(w) ; and $v \to w_1$, $v \to w_2$ implies NUMBER(w_1) > NUMBER(w_2), if (v,w_1) is traversed before (v,w_2) during the second search. Henceforth vertices will be identified according to the numbers assigned to them by the second search. LOWPT1 and LOWPT2 values are recalculated using the new numbering. Two other important sets of numbers are needed. If $u \to v$ in P, let HIGHPT(v) = $\max(\{u\} \cup \{w \mid v \overset{*}{\to} w \;\&\; w \dashrightarrow u\})$. HIGHPT(v) is the highest endpoint of a frond which starts at a descendant of v and ends at the father of v. If v is a vertex, let H(v) be the highest numbered descendant of v. The values of HIGHPT(v) and H(v) for each vertex v are calculated during the second search.

After the second search is completed, we have a palm tree P for G which is ordered according to the adjacency structure A. We also have several sets of numbers associated with the vertices of P. From this information we can determine the biarticulation point pairs of G.

Lemma 1. Let P be a palm tree generated by a depth-first search of a biconnected graph G. Let (a,b) be a biarticulation point pair in G, such that a < b. Then $a \overset{*}{\to} b$ in the spanning tree T of P.

Proof. Suppose that b is not a descendant of a in P. If v is a vertex in T, let D(v) be its set of descendants. The subgraph of G with vertices W = V - D(a) - D(b) is connected. If v is any son of a or b, then the vertices in D(v) are adjacent only to vertices in $D(v) \cup W \cup \{a,b\}$. If (a,b) is a biarticulation point pair, either a or b must be an articulation point, which is impossible since G is biconnected.

An elaboration of Lemma 1 gives a necessary and sufficient condition for (a,b) to be a biarticulation point pair. If $v \to w$ and w is the first entry in the adjacency list of v, then w is called the first son of v. If $v \overset{*}{\to} w$ in P, w is a first descendant of v if each vertex except v on the path $v \overset{*}{\to} w$ is a first son of its father. Every vertex is a first descendant of itself.

Lemma 2. Let P be a palm tree generated by a depth-first search of a biconnected graph G. Let LOWPT1 and LOWPT2 be defined as above. Let (a,b) be a biarticulation point pair in G with $a < b$. Then either:

(1) There are distinct vertices $r \neq a,b$ and $s \neq a,b$ such that $b \rightarrow r$, $\mathrm{LOWPT1}(r) = a$, $\mathrm{LOWPT2}(r) \geq b$, and s is not a descendant of r. (Pair (a,b) is called a biarticulation point pair of type 1.)

Or:

(2) There is a vertex $r \neq b$ such that $a \rightarrow r \overset{*}{\rightarrow} b$; b is a first descendant of r ; $a \neq 1$; every frond $i \dashrightarrow j$ with $r \leq i < b$ has $a \leq j$; and every frond $i \dashrightarrow j$ with $a < j < b$ and $b \rightarrow w \overset{*}{\rightarrow} i$ has $\mathrm{LOWPT1}(w) \geq a$. (Pair (a,b) is called a biarticulation point pair of type 2.)

Conversely, any pair of vertices (a,b) which satisfy either (1) or (2) is a biarticulation point pair.

Proof. The converse part of the lemma is easy to prove. To prove the direct part, suppose (a,b) is a biarticulation point pair in G with $a < b$. By Lemma 1, $a \overset{*}{\rightarrow} b$ in T, the spanning tree of P. Let $b_1, b_2, \ldots, b_n$ be the sons of b in the order they occur in A_b, the adjacency list of b in A. Let $a \rightarrow v \overset{*}{\rightarrow} b$. If w is a vertex in P, let $D(w)$ be the set of descendants of w in T. Let $X = D(v) - D(b)$ and $W = V - D(a)$. If $w \neq v$ is a son of a, some vertex in $D(w)$ is adjacent to some vertex in W, since G is biconnected and vertices in $D(w)$ are adjacent only to vertices in $D(w) \cup \{a\} \cup W$. Vertices in $D(b_i)$ are adjacent only to vertices in $D(b_i) \cup \{a,b\} \cup X \cup W$.

If removal of a and b disconnects some $D(b_i)$ from the rest of the graph, then it is easy to show that (a,b) satisfies (1) with $r = b_i$ and s some vertex in the rest of the graph. If this is not the case, then removal of a and b must disconnect X and possibly some of the $D(b_i)$ from W and the rest of the $D(b_i)$. Furthermore, since

$$\mathrm{LOWPT1}(b_1) \leq \mathrm{LOWPT1}(b_2) \leq \cdots \leq \mathrm{LOWPT1}(b_n) ,$$

there is a $k_0 \geq 1$ such that $W, D(b_1), \ldots, D(b_{k_0})$ are disconnected from $X, D(b_{k_0+1}), \ldots, D(b_n)$. In fact we have

$$k_0 = \max\{i \mid \mathrm{LOWPT1}(b_i) < a\} .$$

Since W and X are not empty, $a \neq 1$ and $v \neq b$. Every frond $i \dashrightarrow j$ with $v \leq i < b$ starts at a vertex in X and hence must satisfy $a \leq j$. Every frond with $a < j < b$ and $b \to b_k \overset{*}{\to} i$ must start in some $D(b_k)$ with $k > k_0$ and hence $LOWPT1(b_k) \geq a$. Because G is biconnected, $LOWPT1(v) < a$. If b were not a first descendant of v, some frond in X would lead to a vertex in W. Thus b must be a first descendant of v, and (a,b) is a biarticulation point pair of type 2, with $r = v$ in the definition of a type 2 pair. This gives the direct part of the theorem. Note that a biarticulation point pair may be both of type 1 and of type 2.

Lemma 2 gives an easy criterion for determining the biarticulation point pairs of G. To test for type 1 pairs, we examine each tree arc $b \to v$ of P and test whether $LOWPT2(v) \geq b$ and either $\neg(LOWPT1(v) \to b)$ or $LOWPT1(v) \neq 1$ or b has more than one son. If so, (LOWPT1(v),b) is a type 1 pair. Testing for type 2 pairs requires a third search. We keep a stack. Each entry on the stack is a triple (h,a,b) of vertices. The triple denotes that (a,b) is a possible type 2 pair and h is the highest vertex which is connected to vertices in D(a) - D(b) by a path which doesn't pass through a or b. (Vertex $h = H(b_{k_0+1})$ where b_{k_0+1} is defined as in the proof of Lemma 2.)

A depth-first search identical to the second search is performed, and the stack of triples is updated in the following manner: When a frond (v,w) is traversed, all triples (h,a,b) on top of the stack with $w < a$ are deleted. If (h_1,a,b_1) is the last triple deleted, a new triple (h_1,w,b_1) is added to the stack. If no triples are deleted, (v,w,v) is added to the stack.

Whenever we return to a vertex $v \neq 1$ along a tree arc $v \to w$ during the search, we test the top triple (h,a,b) on the stack to see if $v = a$. If so, (a,b) is a type 2 pair. We also delete all triples (h,a,b) on top of the stack with $HIGHPT(w) > h$. If w is not the first son of v, let H(w) be the highest descendant of w. We delete all triples (h,a,b) on top of the stack with $H(w) \geq b$. Then we delete all triples with $LOWPT1(w) < a$. If no triples are deleted during the latter step and $\neg(LOWPT1(w) \to v)$, we add the triple (H(w),LOWPT1(w), v) to the stack. Otherwise, if (h,a,b) was the last triple deleted such that $LOWPT1(w) < a$, we add (max{H(w),h}, LOWPT1(w), b) to the stack.

If G has one or more type 2 pairs, we will have discovered one of them when the third search is completed.

Lemma 3. The method described above will find a biarticulation point pair if G is not triconnected. On the other hand, if G is triconnected the method described above will not yield a pair of points.

Proof. If G has a pair of type 1 it will be found by the type 1 test; if G has no type 1 pairs the type 1 test will yield no pairs. This follows from (1) in Lemma 2; a vertex w satisfying (1) exists if and only if either $\neg(\mathrm{LOWPT1}(v) \to b)$ or $\mathrm{LOWPT1}(v) \neq 1$ or b has more than one son.

Suppose now that G has no type 1 pairs. Consider the type 2 test. If (h_1,a_1,b_1) occurs above (h_2,a_2,b_2) in the stack, $a_2 \leq a_1$ and if $a_2 = a_1$ then $b_2 \leq b_1$. Further, if (h,a,b) is deleted from the stack because a frond $v \dashrightarrow w$ is found with $w < a$, then $v < b$. These facts may be proved by induction using the ordering given by A. Every triple (h,a,b) on the stack has $(a \overset{*}{\to} b)$ and a is a proper ancestor of the vertex currently being examined during the search.

If triple (h,a,b) on the stack is tested and it is found that $v = a \neq 1$ when returning along a tree arc $v \to w$, it is straightforward to prove by induction that (a,b) is a type 2 pair. Conversely, if (a,b) is a type 2 pair, let $h = H(b_{k_0+1})$, where b_{k_0+1} is defined in the proof of Lemma 2. Let $a \to v \overset{*}{\to} b$ and let $i \dashrightarrow j$ be the first frond traversed during the search with $v \leq i \leq h$. Then we may prove by induction that (i,j,i) is placed on the stack, possibly modified, and eventually is selected as a type 2 pair. Thus the tests for type 1 and type 2 pairs correctly determine whether G is triconnected.

Lemma 4. The triconnectivity algorithm requires $O(V,E)$ time.

Proof. The three searches, including the auxiliary calculations, require $O(V,E)$ time. Constructing A requires $O(V,E)$ time if a radix sort is used. Testing for type 1 pairs requires $O(V)$ time. Thus the total time required by the algorithm is linear in V and E.

TREE ISOMORPHISM

Suppose we are given two trees T_1 and T_2, and we wish to discover whether T_1 and T_2 are isomorphic. We may assume that T_1 and T_2 are rooted, since if T_1 and T_2 are not rooted it is possible to select a unique distinguished vertex r_i in each tree T_i and call this vertex the root. This is done by eliminating all vertices of degree one (the <u>leaves</u>) from T_i and repeating this step until either a single vertex (the <u>center</u> of T_i) is left or a single edge (whose endpoints are the <u>bicenters</u> of T_i) is left. In the latter case we may add a vertex in the middle of the edge to create a unique root. The time required to determine a unique root in each tree is linear in V, the number of vertices in the trees, if a careful implementation is done.

The difficulty in determining tree isomorphism is that the edges at any vertex have no fixed order. If we could convert each tree into a canonical ordered tree, then testing tree isomorphism would be easy; we merely test for equality of the canonical ordered trees. Thus we need an algorithm which orders the edges at each vertex of the tree. Several authors [Busacker and Saaty (1965A), Lederberg (1964A), Scions (1968A), Weinberg (1965A)] present ordering methods, all virtually the same. Edmond's algorithm [Busacker and Saaty (1965A)] is a good example. Given a rooted tree, the vertices at each level are ordered, starting with those farthest away from the root. After the vertices at level i are ordered, to each vertex at level $i - 1$ is attached a list of its sons (the adjacent vertices at level i).. The vertices at level $i - 1$ are then ordered lexicographically on the lists, according to the order already assigned to the vertices at level i. Once the vertices at each level are ordered, a canonical tree is easy to construct. Neither Edmonds nor any of the other authors who describe this technique note that the ordering process is tricky and must be done carefully if an $O(V)$ time bound is to be achieved.

It is useful to generalize the tree isomorphism problem slightly. We shall allow a set of labels to be attached to each vertex. Each label ℓ must be in the range $1 \leq \ell \leq V$, if V is the number of vertices in the tree. Two labelled trees are isomorphic if they may be matched as unlabelled trees and if any two matched vertices have identical label sets. We shall describe an isomorphism algorithm for labelled trees which requires $O(V,L)$

time, if V is the number of vertices and L the total size of the label sets. All sorting will be done using radix, or bucket sorting, using $2V + 1$ or fewer buckets. Thus the algorithm is suitable for implementation on a random-access computer.

If the trees are unrooted, unique roots are found using the method described above. Level numbers are calculated for all vertices. (The root is level 0.) Next, for each occurrence of a label ℓ an ordered pair (i,ℓ) is constructed; i is the level of this occurrence of ℓ. This set of ordered pairs is sorted lexicographically using two radix sorts to give, for each level i, a list $\mathcal{L}_i$ of the labels (in order) occurring at this level.

Next, we apply Edmond's algorithm, starting at the highest level vertices and working toward the root. Let k be the current level. The vertices at level $k + 1$ will already have been ordered and assigned numbers from 1 to N_{k+1}, where $N_{k+1} \leq V_{k+1}$ and V_{k+1} is the number of vertices at level $k + 1$. Using the list $\mathcal{L}_k$, the labels at level k are changed. The lowest is changed to $N_{k+1} + 1$, the next lowest to $N_{k+1} + 2$, and so on. Next, for each vertex v at level k, an index list containing the numbers of all its sons and all its labels is constructed. The numbers in these lists will be in order if the lists are constructed in the following way: For each son numbered i, an entry is made in its father's index list. This step is repeated for each i in the range $1 \leq i \leq N_{k+1}$. The label numbers are then entered similarly. The sons (the vertices at level $k + 1$) will be in order because of the processing done at level $k + 1$.

Now the vertices at level k must be ordered lexicographically on their index lists. Each number n occurring in an index list is converted into an ordered pair (i,n); i is the position of n in its index list. The set of pairs P is sorted lexicographically using two radix sorts to give a list P_i, for each position i, of the numbers (in order) occurring at that position in the index lists.

Let m be the length of the longest index list. We use m radix sorts, each with $N_{k+1} + |\mathcal{L}_k|$ buckets, to order the index lists lexicographically. During the ith pass, we add to the partially sorted set S of vertices all those whose index list has length $m - i + 1$. We then place vertex $v \in S$ into the buckets according to the value of the $m - i + 1$ th number in the index list of v. The non-empty buckets are emptied in order by referring to the list P_{m-i+1}.

After m passes, the vertices at level k are ordered lexicographically on their index lists. These vertices are numbered from 1 to N_k for some $N_k \leq V_k$; two vertices receive the same number if their index lists are the same. This step completes the processing for level k . After the vertices at each level are ordered, it is easy to construct a canonical ordered tree for each tree, and testing isomorphism is then a simple equality test.

Finding a root for each tree requires $O(V)$ time. Constructing the label lists $\mathcal{L}_i$ requires $O(L)$ time. Constructing the index lists at level k requires $O(V_{k+1} + |\mathcal{L}_k|)$ time. Ordering the vertices at level k lexicographically on their index lists requires m passes but only $O(V_{k+1} + |\mathcal{L}_k|)$ time, because only nonempty buckets are emptied on each pass and the total number of entries made in the buckets is $V_{k+1} + |\mathcal{L}_k|$. Since $\sum_k (V_k + |\mathcal{L}_k|) \leq V + L$, the entire algorithm has an $O(V,L)$ time bound. If the trees are unlabelled, or if each tree has at most one label, the tree isomorphism algorithm requires $O(V)$ time.

ISOMORPHISM OF TRIPLY CONNECTED PLANAR GRAPHS

In this section we describe an algorithm for partitioning a set of triply connected planar graphs into subsets of isomorphic graphs. The asymptotic running time of the algorithm grows as $V \log V$ where V is the total number of vertices in all graphs. An algorithm for such a partitioning [Hopcroft (1971)] based on converting the graphs to finite automata was previously used. However, working directly with graphs leads to major simplifications since the finite automata which are obtained from conversion of planar graphs are a very restricted subset of all finite automata and the full power to partition arbitrary finite automata is not needed.

Let G be a planar graph whose connected components are triply connected pieces other than n-gons or n-bonds . Consider a fixed embedding of G in the plane. We treat each edge of G as two directed edges. Let (v_1,v_2) be a directed edge. We denote (v_2,v_1) by $(v_1,v_2)^r$. We write $(v_1,v_2) \overset{R}{\vdash} (v_2,v_3)$ and $(v_1,v_2) \overset{L}{\vdash} (v_2,v_4)$ where (v_2,v_3) and (v_2,v_4) are edges bounding the faces to the right and left, respectively, of (v_1,v_2) .

Let ε denote the string of length zero. For each edge e we write $e \overset{\varepsilon}{\vdash} e$. Let x be a string consisting of R's and L's . If $e_1 \overset{x}{\vdash} e_2$ and $e_2 \overset{R}{\vdash} e_3$ (or $e_2 \overset{L}{\vdash} e_3$) we write $e_1 \overset{xR}{\vdash} e_3$ (or $e_1 \overset{xL}{\vdash} e_3$). We write $e_1 \vdash e_2$ if x is understood. Intuitively we write $e_1 \overset{x}{\vdash} e_2$ if e_2 is reached by starting at e_1 and and traversing a path in the graph dictated by x . Each symbol of x dictates which way to turn on entering a vertex. On entering a vertex by edge e , leave the vertex by the edge immediately to the right or left of the edge e depending on whether the corresponding symbol of x is R or L respectively. Note that $e_1 \overset{*}{\Rightarrow} e_2$ does not necessarily imply $e_1 \vdash e_2$ since $e_1 \overset{*}{\Rightarrow} e_2$ denotes an arbitrary path and $e_1 \vdash e_2$ denotes a special type of path. Let λ be a mapping of edges into the integers such that $\lambda(e_1) = \lambda(e_2)$ if and only if the number of edges on the face to the right (left) of e_1 is the same as the number of edges on the face to the right (left) of e_2 , the degrees of the heads of e_1 and e_2 are the same, and the degrees of the tails of e_1 and e_2 are the same.

Lemma 5. Let e be a directed edge and let v be a vertex in the connected component containing e.

1) There exists an edge e_1 directed into v such that $e \vdash e_1$.
2) Let e_1 be a directed edge into v. If v is of odd degree $e \vdash e_1$. If v is of even degree either $e \vdash e_1$ or $e \vdash e_1^r$.

Proof. (1) It suffices to show that for any v adjacent to the head of e there exists an edge e' directed into v such that $e \vdash e'$. Let $e_1, e_2, \ldots, e_m$ be the edges directed into the head of e. Consider the path e_1, e_2^r, followed by edges around the face to the left of e_2^r until e_3, e_4^r, followed by edges around the face to the left of e_4 and so on back to e_1. This path enters every vertex adjacent to v.

(2) Follow path to v and then twice around v by a path similar to that in (1).

Lemma 6. Let e_1, e_2, e_3 and e_4 be directed edges.

1) If $e_1 \vdash e_2$ then $e_2 \vdash e_1$ and $e_1^r \vdash e_2^r$.
2) If $e_1 \vdash e_1^r$ then $e_1 \vdash e_2$.
3) Assume there exists a path $p_1 \colon e_1 \overset{*}{\Rightarrow} e_3$. Further assume that there exists a corresponding path $p_2 \colon e_2 \overset{*}{\Rightarrow} e_4$. By corresponding we mean that both paths are the same length, that the number of edges clockwise between the edge into and the edge out of corresponding vertices agree, and that corresponding edges have the same value of λ. Then either
   (a) $e_1 \vdash e_3$ and $e_2 \vdash e_4$ or
   (b) $e_1 \vdash e_3^r$ and $e_2 \vdash e_4^r$ by the same sequence of right and left turns.

Proof. (1) Consider the path p from e_1 to e_2. Let e be the next-to-last edge in p. Without loss of generality we can assume e_2 bounds the face to the right of e and we need only show $e_2 \vdash e$. Clearly $e_2 \vdash e$ by a path which travels clockwise around the face to the right of e_2. Having established that $e_2 \vdash e_1$ it immediately follows that $e_1^r \vdash e_2^r$ by reversing all edges on the path $e_2 \vdash e_1$.

(2) By Lemma 5 either $e_1 \vdash e_2$ or $e_1 \vdash e_2^r$. If $e_1 \vdash e_2^r$ then by part 1 of Lemma 6 $e_1^r \vdash e_2$.

(3) By the construction in Lemma 5 either $e_1 \vdash e_3$ or $e_1 \vdash e_3^r$ by a path only using edges bounding faces adjacent to p_1 . A corresponding construction using p_2 yields the desired result.

Two edges e_1 and e_2 are said to be <u>distinguishable</u> if and only if there exists a string x such that $e_1 \overset{x}{\vdash} e_3$, $e_2 \overset{x}{\vdash} e_4$ and $\lambda(e_3) \neq \lambda(e_4)$. If e_1 and e_2 are not distinguishable they are said to be <u>indistinguishable</u>.

We need the following technical lemma.

<u>Lemma 7</u>. Let G be a biconnected planar graph. Let $(v_1,v_2)(v_2,v_3),\ldots,(v_{n-1},v_n)$ be a simple path p in G . Then there exists a face having an edge in common with the path which has the property that the set of all edges common to both the face and the path form a continuous segment of the path. Furthermore, when traversing an edge of the face while going from v_1 to v_n along the path, the face will be on the right.

<u>Proof</u>. See [Hopcroft (1971A)].

<u>Theorem 8</u>. Edges e and e' are indistinguishable if and only if there exists an isomorphism of the embedded version of G which maps e onto e' .

<u>Proof</u>. The if portion of the theorem is obvious. Namely, if e is mapped to e' by some isomorphism, then it is easily seen that e and e' are indistinguishable. The only if portion is more difficult to prove and we first establish it for regular degree three graphs.

Let G be regular of degree 3 and assume that edges e and e' are indistinguishable. We will now exhibit a method of constructing an isomorphism identifying e and e' . If e and e' are in the same connected component, then each edge not in the component is mapped to itself. If e and e' are in different components, say C_1 and C_2 , then each edge not in C_1 or C_2 is mapped to itself.

Identify edge e with e' . When two edges are identified, their reversals and their corresponding heads and tails are automatically identified. Whenever an edge e_1 is identified with an edge e_2 , identify edges e_3 and e_4 where $e_1 \overset{R}{\vdash} e_3$ and $e_2 \overset{R}{\vdash} e_4$. If e_3 has already been identified with e_4 , then select some pair of edges e_5 and e_6 which have already been identified, while e_7 and e_8 have not, where $e_5 \overset{L}{\vdash} e_7$ and $e_6 \overset{L}{\vdash} e_8$. Identify e_7 and e_8 and repeat the process. In other words always use the symbol R to obtain new edges, if possible, otherwise use L . This means that we will always identify edges along a path until we reach a pair of vertices already identified.

By Lemma 5(part 2), this procedure will yield the desired isomorphism unless a conflict arises. A conflict arises when we try to identify a vertex v_1 with a vertex v_2 which has already been identified with some $v_3 \neq v_1$. We now prove that such a situation is impossible.

Assume a conflict arises and consider the first such instance. One of the edges in the last pair identified must have completed a cycle. It is important to note that an edge that completes a cycle must terminate at a vertex which has both of the other edges already identified. The corresponding edge either did not complete a cycle (i.e. it terminated at an unidentified vertex) or it completed a different cycle (the end vertices of the two edges had previously been identified but not with each other). In the latter case the cycles are of different lengths. If both edges completed cycles, let c be the shorter of the two cycles. If only one cycle is completed, let c be that cycle. Let p be the path in the other graph corresponding to the vertices on the cycle c . The first and last vertices of p correspond to the same vertex c.

Since there is a cycle which is mapped to a simple path, select that cycle c which would map to a simple path but for which no cycle other than c containing only vertices from c and its interior would map to a simple path. By Lemma 7 some face is adjacent to p on the right and all edges of the face which are common to p form a continuous segment of p . Start identifying the edges around this face with edges on the interior of c . One of three cases occurs. (1) We return to a vertex on p before returning to a vertex on c , (2) We return to a vertex on c before returning to a vertex on p , or (3) Both events occur simultaneously. If (1) occurs, a face is identified with a non-closed path. This is impossible since λ contains information as to the number of edges around the face to the right or left of each edge. (2) is impossible since no cycle on the interior of c maps to a non-closed path. If (3) occurs we must have identified corresponding faces. This implies that the paths terminated at corresponding vertices and that c has been divided into two **cycles** c_1 and c_2. Assume c_1 corresponds to the face. Then cycle c_2 is mapped to a path, a contradiction. Since all possibilities lead to a contradiction, we are forced to conclude that no conflict can arise.

Having established the theorem for the special case where G is regular of degree 3, we now prove the theorem in general. Let G be an embedding of a planar graph all of whose connected components are triply connected. Assume that the edges e_1 and e_2 are indistinguishable but that there is no isomorphism mapping e_1 onto e_2. Let $\hat{G}$ be the graph obtained by expanding each vertex of degree $d > 3$ into a d-gon. Let $\hat{e}_1$ and $\hat{e}_2$ be the edges of $\hat{G}$ corresponding to e_1 and e_2. Since $\hat{G}$ is regular of degree 3, $\hat{e}_1$ and $\hat{e}_2$ must be distinguishable. Let $\hat{p}$ be the path that distinguishes $\hat{e}_1$ and $\hat{e}_2$. Clearly there is a corresponding path in G. By Lemma 6 (part 3) there exists an x which distinguishes e_1 and e_2, a contradiction.

The v log v isomorphism algorithm depends on an efficient algorithm for partitioning the edges of a planar graph into sets so that two edges are placed in the same set if and only if they are indistinguishable. This is done as follows.

Initially the edges are partitioned so that e_1 and e_2 are in the same set if and only if $\lambda(e_1) = \lambda(e_2)$. The index of each block of the partition except one is placed on a list called Rightlist and on a list called Leftlist. Let $B_1, B_2, \ldots, B_i$ be the current blocks of the partition. The blocks of the partition are refined by applying the following procedure until the Rightlist and Leftlist are empty.

Select the index j of some block from Rightlist or Leftlist and delete it from the list. Assume it came from Rightlist. For each e in B_j mark the edge e' defined by $e \overset{R}{\vdash} e'$. If block B_i contains both marked edges and unmarked edges partition it into two blocks B_{i_1} and B_{i_2} so that one contains only marked edges, the other only unmarked edges. Assume B_i is partitioned into blocks B_{i_1} and B_{i_2} where $|B_{i_1}| \leq |B_{i_2}|$. If the index i is already on Rightlist, replace it by i_1 and i_2; otherwise add only i_1 to Rightlist. Similarly if the index i is already on Leftlist, replace it by i_1 and i_2; otherwise add only i_1 to Leftlist.

Theorem 9. The above algorithm terminates and on termination two edges are in the same B_i if and only if they are indistinguishable.

Proof. (if) In the initialization phase edges are placed in different blocks only if they are immediately distinguishable. Subsequently a block is partitioned with e_1 and e_2 going into different blocks only if there exist e_3 and e_4 already in different blocks with $e_3 \overset{R}{\vdash} e_1$ and $e_4 \overset{R}{\vdash} e_2$ or $e_3 \overset{L}{\vdash} e_1$ and $e_4 \overset{L}{\vdash} e_2$. In the former case $e_1{}^r \overset{L}{\vdash} e_3{}^r$ and $e_2{}^r \overset{L}{\vdash} e_4{}^r$, and in the latter case $e_1{}^r \overset{R}{\vdash} e_3{}^r$ and $e_2{}^r \overset{L}{\vdash} e_4{}^r$. In either case e_1 and e_2 are distinguishable.

(only if) Assume e_1 and e_2 are distinguishable. We will prove that e_1 and e_2 are placed in different blocks of the partition by induction on the length n of the shortest string distinguishing $e_1{}^r$ and $e_2{}^r$. Assume the induction hypothesis is true for sequences of length n and that the shortest sequence distinguishing $e_1{}^r$ and $e_2{}^r$ is of length $n+1$. Without loss of generality assume the sequence is xR. Then x is of length n and distinguishes some $e_3{}^r$ and $e_4{}^r$ where $e_3 \overset{R}{\vdash} e_1$ and $e_4 \overset{R}{\vdash} e_2$. By the induction hypothesis e_3 and e_4 will be placed in separate blocks. The index of one of these blocks will go onto Rightlist and when it is removed e_1 and e_2 must be placed in separate blocks.

Theorem 10. The running time of the partitioning algorithm is bounded by $kV \log V$ for some k.

Proof. Clearly the running time of the algorithm is dominated by the time spent in partitioning blocks. Assume that at some point the blocks of the partition are $B_1, B_2, \ldots, B_m$. Let b_i be the cardinality of B_i . Let M be the set of integers from 1 to M , let I be the indices on Rightlist and let J be the indices on Leftlist. We claim that the time remaining is bounded by

$$T = k(\sum_{i\varepsilon I} b_i \log b_i + \sum_{i\varepsilon M-I} b_i \log (b_i/2) + \sum_{i\varepsilon J} b_i \log b_i$$
$$+ \sum_{i\varepsilon M-J} b_i \log(b_i/2))$$

Clearly the bound holds when the algorithm has terminated. Consider what happens when an index i_0 is selected from I . Certain blocks will be partitioned. The remaining time will be bounded by some new T' . The time spent in partitioning is bounded by kb_{i_0} . We must show that

$$kb_{i_0} + T' \leq T .$$

Assume a block of size b_i is partitioned into blocks of size c_i and b_i-c_i where $c_i \leq b_i/2$. Clearly

$$b_i \log b_i \geq c_i \log c_i + (b_i-c_i)\log(b_i-c_i) \text{ and}$$
$$b_i \log b_i/2 \geq c_i \log c_i/2 + (b_i-c_i)\log(b_i-c_i)/2 .$$

Thus we need only show that

$kb_{i_0} + kb_{i_0}\log(b_{i_0}/2) \leq k\, b_{i_0}\log b_{i_0}$. This follows since $b_{i_0} + b_{i_0}\log b_{i_0}/2 = b_{i_0}(1 + \log b_{i_0}/2) = b_{i_0}\log b_{i_0}$.

This completes the proof.

The partitioning algorithm can be used to partition a set of triply connected planar graphs into subsets of isomorphic graphs. Each triply connected planar graph has exactly two embeddings in the plane. One of the embeddings is obtained from the other by reversing the order of all edges around each vertex. We will refer to these embeddings as the clockwise and counterclockwise embeddings. Given a collection $G_1, G_2, \ldots, G_n$ of triply connected planar graphs, a composite graph G is formed consisting of two copies of each G_i . G is embedded in the plane so that one copy of each G_i has the clockwise embedding and one copy of each G_i has the counterclockwise embedding. The partitioning algorithm is applied to the embedding of G . As a consequence of

Theorems 8 and 9, G_i is isomorphic to G_j iff for any edge e in an embedding of G_i there exists an edge e' in one of the embeddings of G_j such that e and e' are in the same block of the partitioning of the edges. We can now easily partition the G_i into isomorphic graphs as follows: Place G_1 in the first block of the partition. Select an edge e of G_1. Scan the block of the edge partition containing e. For each j such that either embedding of G_j has an edge in the same block of the edge partition place G_j into the block containing G_1. Select the smallest k such that G_k is not already placed in Block 1 and place G_k in Block 2. Select an edge e of G_k and scan the block of the edge partition containing e as before. The whole process is repeated until every G_k is placed into some block.

The time necessary to complete the above process is bounded as follows: First the planar embeddings of the G_i must be determined. For each G_i the time required is proportional to the number of vertices in G_i [Tarjan (1972B)]. Thus the total time for this step is proportional to the number of vertices in the composite graph G. The time for the edge partitioning is bounded by kE log E where k is some constant and E is the number of edges in the composite graph G. Thus the total time is bounded by some constant times E log E. Since in a planar graph $E \leq 3V-6$, the total time is $O(V \log V)$, where V is the number of vertices in G.

ISOMORPHISM ALGORITHM

Given two graphs G_1 and G_2, the isomorphism algorithm divides the graphs into connected components, subdivides each connected component into biconnected components and then further subdivides the biconnected components into triply connected components. The connectivity structure of each graph is represented as follows: Each graph is represented by a tree consisting of a root plus one vertex for each connected component. Each connected component is represented by a tree consisting of one vertex for each biconnected component and one vertex for each articulation point.

Let v_a be a vertex corresponding to an articulation point a and let v_B be a vertex corresponding to a biconnected component B containing a. Then v_a and v_B are connected by an edge. It is easy to see that the resulting graph is indeed a tree. The leaves of a tree representing a connected component are called <u>2-leaves</u>.

Each biconnected component B is also represented by a tree. The set of vertices consists of one vertex for each biarticulation point pair and one vertex for each triply connected component. If v_C represents a triply connected component C and v_{ab} represents a biarticulation point pair (a,b) then an edge connects v_C and v_{ab} if and only if a and b are both contained in C. The leaves of a tree representing a triply connected component are called <u>3-leaves</u>.

Consider the trees corresponding to connected components. Each 2-leaf is a biconnected component which corresponds to a tree-like structure of triply connected components. All triply connected components which are 3-leaves and are contained in the 2-leaves are assigned an ordered pair of numerical codes, such that equality of codes is equivalent to isomorphism. This is done by a method described below. These triply connected components are deleted and their codes are attached to new edges joining their biarticulation points in the remaining part of the graphs. This process creates new 3-leaves within the 2-leaves. These new 3-leaves are found and codes generated for them, and the process is repeated until each biconnected component which is a 2-leaf is reduced to a single edge. These edges are deleted, their codes being attached to the corresponding articulation points in the remaining part of the graphs. The new biconnected components which are 2-leaves are found and the process is repeated. Eventually, each connected component is reduced to a single vertex with an attached isomorphism code. The codes for the components of each graph are sorted and compared for equality. If they are equal, the graphs are isomorphic; if not, the graphs are non-isomorphic.

Consider any 3-leaf. Note that it has an orientation with respect to its biarticulation points. It is for this reason that we assign an ordered pair of numbers to the 3-leaf; the numbers are equal if and only if the 3-leaf is symmetric with respect to an exchange of the biarticulation points. In order to assign pairs of integers to a set of 3-leaves, the components are tested for planarity by a linear algorithm [Hopcroft and Tarjan (1972A), Tarjan (1972B)]. If any component is not planar, the isomorphism algorithm halts, since the entire graph is not planar. Assuming all 3-leaves are planar, the planarity algorithm constructs a planar representation of the component which is essentially unique since the component is triply connected. The $O(V \log V)$ algorithm described in Section 4 is used to determine the equivalence classes of isomorphic 3-leaves. Ad hoc integer codes are assigned to the biarticulation points of each 3-leaf according to the equivalence classes, in such a way that isomorphic components are assigned identical ordered pairs of integers and non-isomorphic components are assigned different ordered pairs.

The running time of the algorithm is dominated by the time required to partition the triply connected components into equivalence classes of isomorphic graphs. If V is the total number of vertices in each of the original graphs, then the partitioning requires $O(V \log V)$ time. The time to find biconnected and triply connected components, to test all 3-leaves for planarity, and to construct a planar representation, is $O(V)$; which is dominated by $O(V \log V)$.