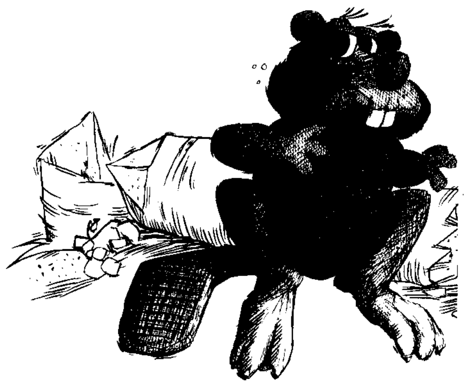


Linear Time Automorphism Algorithms for Trees, Interval Graphs, and Planar Graphs

Charles J. Colbourn
Kellogg S. Booth

CS-79-06

March, 1980

Linear Time Automorphism Algorithms for Trees, Interval Graphs, and Planar Graphs

Charles J. Colbourn

Department of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 1A7

Kellogg S. Booth

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

ABSTRACT

An algorithm based upon Edmonds's procedure for testing isomorphism of trees is extended to answer various questions concerning automorphisms of a labeled forest. This and linear pattern matching techniques are used to build efficient algorithms which find the automorphism partition and a set of generators for the automorphism group, determine the order of the automorphism group, and compute a coding for forests, interval graphs, outerplanar graphs, and planar graphs.

This material has been submitted for publication in *SIAM Journal on Computing*. The work was supported in part by the Natural Sciences and Engineering Research Council of Canada under Grant No. A4307.

1. Preliminaries

Starting from an algorithm for tree isomorphism we show how to build efficient algorithms which can answer a number of questions concerning the automorphisms of labeled forests. These algorithms operate in time which is linear with the size of their input. They can be applied to other types of graphs whenever those graphs can be conveniently represented by labeled forests. This is the case for interval graphs and for planar graphs, as will be demonstrated in later sections. Applying linear pattern matching techniques and a knowledge of their additional structure we can produce an even faster algorithm for outerplanar graphs, a special case of planar graphs which has been examined in the literature.

We assume the usual definitions used in graph theory [3,11]. Let $G = (V,E)$ and $G' = (V',E')$ be two graphs. A bijection $\phi : V \rightarrow V'$ which maps pairs of adjacent vertices onto pairs of adjacent vertices and pairs of nonadjacent vertices onto pairs of nonadjacent vertices is an *isomorphism* of G with G'. An *automorphism* of a graph G is an isomorphism of G with itself. The *automorphism group* of G is the group whose elements are the automorphisms of G. A *set of generators* for the automorphism group of G is any set of automorphisms whose closure under functional composition and inverse is the entire automorphism group. Two vertices x and y are *similar* in G whenever there exists an automorphism of G which carries x onto y. Similarity is an equivalence relation on V whose equivalence classes form the *automorphism partition* of G. A *coding* is a function on graphs such that the values assigned to G and G' are identical if and only if G and G' are isomorphic.

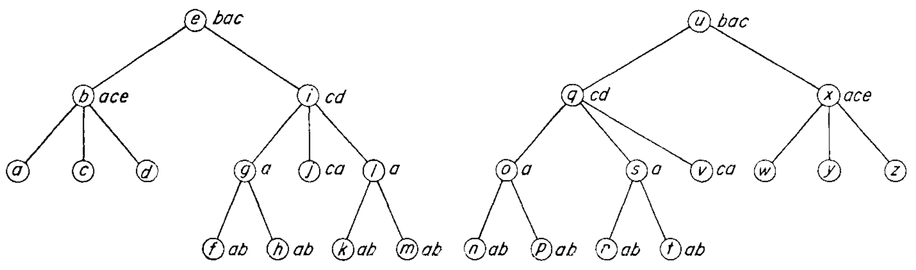
In the following sections we will first review the tree isomorphism test and then extend it to answer other questions about automorphisms of a labeled forest. Using these results as a foundation we will go on to construct efficient automorphism algorithms for interval graphs and outerplanar graphs. Finally, we will sketch how algorithms for planar graphs might be build using these same ideas.

Weinberg previously presented algorithms for computing the automorphism partitions of triconnected planar graphs and of trees. His algorithms require $O(n^2)$ steps [36,37]. Corneil, and later James and Riha, have produced substantially simpler algorithms for the tree problem. Their algorithms also have an $O(n^2)$ time complexity [8,21]. Most recently Fontet has given an $O(n)$ algorithm for arbitrary planar graphs which is very similar to the approach taken here [9]. To our knowledge there are no linear time algorithms in the literature for finding a set of generators for the automorphism group of a planar graph or for computing the order of its automorphism group. All of the results on automorphisms of interval graphs and outerplanar graphs are new. Additional references to previous work are contained within subsequent sections.

2. Trees

A *tree* is a connected graph having no cycles. Edmonds presented an efficient procedure for testing isomorphism of trees which is based on a canonical numbering of the vertices [6]. This method has been rediscovered a number of times by different researchers [22,30,34]. A description of the algorithm and a proof that it runs in linear time is given by Aho, Hopcroft, and Ullman [1, Section 3.2]. Their version of the algorithm allows the trees to be labeled but assumes that the trees are rooted. This is no real drawback because unrooted trees can be rooted in a unique fashion by finding the center or bicenter of the tree and using it as a root. This can easily be accomplished in linear time. For our purposes we will assume that all trees have been rooted.

These techniques can be applied to forests of trees having labels which consist of integers or strings of integers in a range which is $O(n)$ where n is the number of vertices in the forest. All of our algorithms will be linear in the size of the forest plus the sum of the lengths of the labels. The forest algorithms will then become the building blocks for the algorithms in later sections.



Rank	1	2	3	4	5	6	7
Label	Λ	*a*	*ab*	*ace*	*bac*	*ca*	*cd*

FIGURE 2.1: An example of a labeled forest. Vertex names (symbols within vertices) are used to identify vertices and should not be confused with vertex labels or with any of the numbers which will later be assigned to vertices. Rank information for the bucket sorted labels is displayed below the forest. These ranks will be used to generate working labels for the *i*-numbering which follows.

We briefly sketch our version of Edmonds's procedure for testing isomorphism of two labeled forests. All of the labels are first bucket sorted and assigned an integer rank within the sorted list. Then each vertex in the forest receives an integer *i-number* according to the following scheme. Beginning at the vertices of maximum depth in the forest a *working label* is assigned which consists of the rank of the original label followed by the *i*-numbers, in increasing order, of all the children. The working labels of all of the vertices at the current depth are then sorted and the *i*-number of each vertex becomes the rank of its working label within the sorted list of working labels. This process proceeds upwards until the roots of all of the trees have received an *i*-number. Using a bucket sort it is possible to perform all of these operations so that the total work is linear in the size of the forest plus the sum of the lengths of the original labels [1].

A complete isomorphism test consists of performing the *i*-numbering on two labeled forests in parallel, checking at each level that the sets of working labels are the same in both forests. If a mismatch occurs at any level the forests cannot be

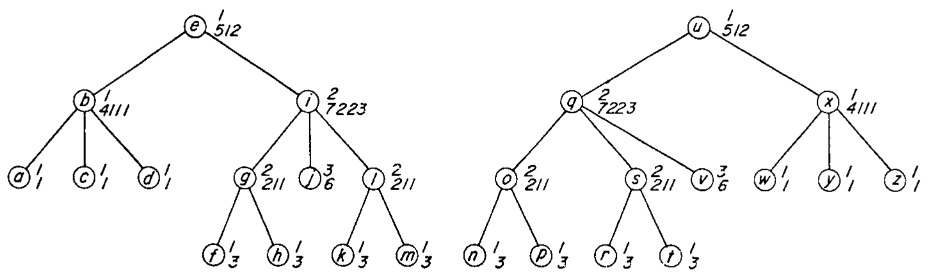


FIGURE 2.2: The *i*-numbering for the forest in Figure 2.1. Shown to the right of each vertex are its *i*-number (above) and its working label (below). A working label consists of the rank of the original vertex label followed by the *i*-numbers (in increasing order) of the children. Note that because the *i*-numbers of the roots are identical the two trees must be isomorphic.

isomorphic and the algorithm terminates announcing non-isomorphism. Otherwise the algorithm succeeds in finding an isomorphism when the sets of *i*-numbers assigned to the roots are identical in the two forests.

We can use a similar procedure to get other information concerning the automorphisms of a forest. Only a little more work is necessary to find the automorphism partition. Suppose that the forest has been *i*-numbered. The vertices are partitioned into equivalence classes by their *i*-numbers together with their depths. This is usually not the automorphism partition but the automorphism partition is always a refinement of this partition and can be determined from a second pass over the forest, this time top down.

LEMMA: 2.1: Let G be an *i*-numbered labeled forest. Two vertices x and y are similar if and only if they have the same depth and *i*-number and their parents (if present) are similar.

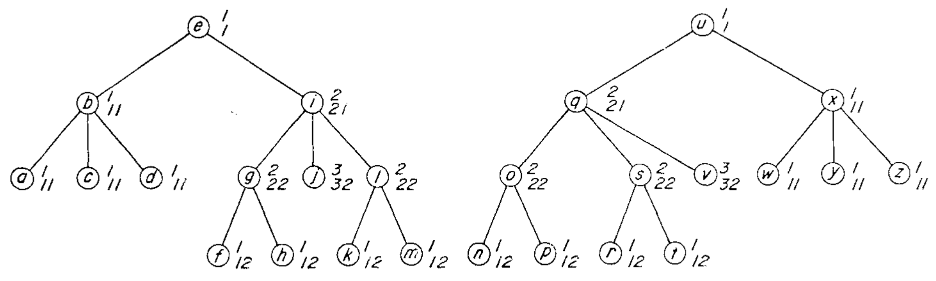
PROOF: *Only If:* Let x and y be two similar vertices in G. If $x = y$ then the conditions hold trivially. So assume that $x \neq y$. Any automorphism must preserve depths and ancestors within a forest. Thus x and y are at the same depth and either have similar parents or no parents at all. It remains only to show that their *i*-numbers are the same.

From the correctness of the isomorphism algorithm which generates the *i*-numbers it follows that two vertices at the same depth have the same *i*-number if and only if the subtrees rooted at those vertices are isomorphic. Since x is similar to y their subtrees are isomorphic and hence their *i*-numbers are indeed identical.

If: Again we assume that $x \neq y$ but that both are at the same depth, have similar parents, and the same *i*-numbers. If x and y are at depth 0 they are roots in the forest and by the observation cited before their trees are isomorphic and hence they must be similar vertices. Otherwise x and y are at a positive depth and both have parents which are similar within the forest. We show that there is an automorphism carrying x onto y and thus complete the proof.

Any automorphism mapping the parent of x to the parent of y must map x to some sibling of y. Such an automorphism exists (the parents are similar) so we can choose some sibling of y. Call this sibling z. Clearly x and z are similar. Using the only-if part of this lemma x, y, and z all have the same i-number and thus the subtrees rooted at y and z must be isomorphic. We produce the desired mapping by composing the automorphism which carries x to z with the automorphism which interchanges the two siblings y and z. This mapping is an automorphism of G and it carries x onto y. □

We now explain how to compute the automorphism partition of a labeled forest. After rooting each tree at its center or bicenter we assign i-numbers as before. A second set of integer *j-numbers* is then assigned in a top down fashion. At each level beginning with the roots, vertices are assigned working labels which consist of the i-number of the vertex followed by the j-number of the parent. These working labels are then sorted on each level and the j-number for the vertex is the rank of the working label within the sorted list of working labels. From the lemma it follows that the j-numbers plus depth information partition the vertices into similarity classes.



The blocks of the automorphism partition for the forest

$\{a,c,d,w,y,z\}$ $\{b,x\}$ $\{e,u\}$ $\{f,h,k,m,n,p,r,t\}$ $\{g,l,o,s\}$ $\{i,q\}$ $\{j,v\}$

FIGURE 2.3: The forest from Figure 2.1. Shown to the right of each vertex are its j-number (above) and its working label (below). A working label consists of the i-number of the vertex followed by the j-number of its parent. The working label for a root is its i-number. The seven blocks of the automorphism partition are shown below the forest. These are found by making all vertices which have the same depth and j-number equivalent.

This j-numbering algorithm is easily implemented to run in linear time. It is a straightforward extension of Edmonds's algorithm. We have now shown that the automorphism partition of a labeled forest can be found in linear time. So far our work duplicates results of Fontet who also realized that Edmonds's approach could be used to compute the automorphism partition of a tree [9]. We go further, however, and produce a set of generators for the automorphism group and also determine the order of the automorphism group.

Let us first consider the problem of computing the automorphism group of a forest. We cannot hope to achieve an algorithm whose running time is linear in the size of the forest. The size of the output alone prohibits this. The star $K_{1,n}$, for example, has $n!$ automorphisms. Even computing a set of generators for such a group is costly. Weinberg has shown how to compute a set of generators in $O(n^2)$ time and space. The time bound cannot be improved because all sets of generators could contain $\Omega(n)$ automorphisms and each generator has $\Omega(n)$ ordered pairs.

Since there is no way to circumvent this problem we can instead agree to a *compact representation* for the automorphisms. If we represent an automorphism ϕ by its set of non-fixed points then ϕ becomes the set

$$\{(x, \phi(x)) \mid x \neq \phi(x)\},$$

and we can achieve an $O(n)$ algorithm which produces a set of generators for the automorphism group of a labeled forest.

LEMMA: 2.2: The compact representation of a set of generators for a labeled forest can be computed in linear time.

PROOF: First compute the automorphism partition of the forest. From each similarity class of roots choose a single representative r. For each of the other roots r' which are similar to r output the compact representation of the automorphism which maps the tree rooted at r onto the tree rooted at r', all other vertices within the forest remaining fixed. This can be accomplished by walking the two trees in parallel. Discard all of the trees rooted at the various r' keeping only the tree rooted at r.

After all of the similarity classes of roots have been processed delete the roots from the remaining forest and repeat this process level-by-level until the forest is empty. The time and space used is $O(n)$ because each vertex appears at most once in the range of an automorphism. A detailed proof that these mappings are actually automorphisms and that they generate the entire group is given in the first author's masters thesis [7]. □

The final automorphism problem is handled in a like manner. Suppose that we want to determine the order of the automorphism group. Mathon [24] has shown that in general this problem is polynomially equivalent to testing graph isomorphism, but his reduction involves a factor of n^2 increase in running time. We will show that for forests (and hence trees) the order of the automorphism group can be computed in linear time.

Given a j-numbered forest the order of the automorphism group is found by a third walk of the forest, this time bottom up as in the original i-numbering. This pass generates *k-numbers* which indicate the order of the automorphism group for the subtree rooted at each vertex.

LEMMA: 2.3: Let G be a rooted j-numbered forest in which x_0 is a vertex and x_1, x_2, ... , x_p are its children. If the p children are partitioned into q classes according to their j-numbers and c_t is the size of class t then the number of automorphisms for the subtree rooted at x_0 is given by

$$k_0 = \left(\prod_{s=1}^{p} k_s \right) \cdot \left(\prod_{t=1}^{q} c_t! \right)$$

where k_s is the number of automorphisms for the subtree rooted at x_s, for $0 \leqslant s \leqslant p$.

PROOF: Any automorphism of the subtree rooted at x_0 must carry each child of x_0 to a similar child of x_0. For each of the q classes there are exactly $c_t!$ ways of permuting the c_t children in that class. In addition the subtree rooted at each x_s has k_s automorphisms. All of the choices are independent so the total number of possibilities is the product of these numbers. □

An algorithm to compute the order of the automorphism group of a forest is now easy. Working from the bottom up each vertex receives its k-number by counting the number of children within each similarity class (using the j-numbers) and then multiplying together the appropriate factorials and the k-numbers of the children. If an imaginary root is added acting as a parent for all of the real roots in the forest then its k-number is precisely the order of the automorphism group for the forest.

The entire calculation is linear with the exception of the factorial computation. If factorial is not a primitive operation in the model of computation a table of factorials for the integers from 0 to n can be precomputed and stored in a table of size $O(n)$. The cost of building the table is $O(n)$ if a uniform cost measure is used. Under the more realistic logarithmic cost criterion, in which the number of bits involved in each operation is counted, the cost becomes $O(n^2 \log n)$ since $n!$ requires $\Theta(n \log n)$ bits [1].

This last observation effectively kills any hope of a linear algorithm since a tree whose automorphism group has anywhere near $n!$ elements will require output whose representation is too large to print in linear time. Our point here is that the number of "data" operations need only be linear in the size of trees being considered; we ignore the fact that pointers and indices require $\Theta(\log n)$ bits for a graph on n vertices and that arithmetic operations on large numbers will require more than unit cost. Even for problems of practical size $n!$ is large and multiple precision arithmetic will have to be used, although if only an order of magnitude result is required we could use a floating-point calculation (or its equivalent, a fixed-point approximation of the logarithm). Asymptotically this may be an unrealistic assumption but we ignore this point and maintain the fiction that arbitrary integers can be held in a single register, adopt the uniform cost criterion, but keep in mind the warning made by a referee that "[this] assumption – *if abused* – leads to all sorts of unexpected consequences."

Being able to compute the order of the automorphism group enables us to answer additional questions about isomorphisms. In particular the number of isomorphisms between two forests is easy to compute. Mathon has noted that any two graphs either are non-isomorphic or else their number of isomorphisms between them is the same as the number of automorphisms of one of the graphs [24]. Our algorithm is easily adapted to this purpose. We omit the details here.

We recall in passing that Edmonds's algorithm is already known to provide an efficient method for coding trees. The modification for forests is trivial. Starting with a sorted list of the i-numbers for the roots we recursively substitute the working label, in parentheses, for each vertex's i-number (also substituting original labels for ranks) until we have a single string consisting of the original labels suitably parenthesized to indicate the structure of the forest. This is a

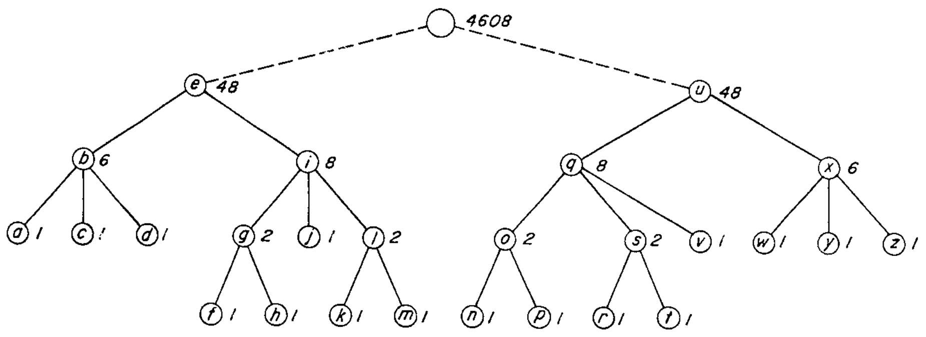


FIGURE 2.4: The k-numbers for the forest in Figure 2.1. An imaginary root vertex has been added to the forest. Its k-number is the order of the automorphism group, in this example 4608.

canonical representation. The code for each tree in our example forest is the string

$$(bac\,(ace\,()()())(cd\,(a\,(ab\,)(ab\,))(a\,(ab\,)(ab\,))(ca\,)))$$

and the code for the entire forest is the tree code repeated twice.

In summary we can state the following result about the automorphisms of a labeled forest. As a special case these same results apply to trees.

THEOREM: 2.4: It is possible to test isomorphism, to find the automorphism partition or a set of generators for the automorphism group, and to count the number of automorphisms or isomorphisms for labeled forests in time and space which is linear in the size of the forests plus the sum of the lengths of the labels.

We will use these results in the following sections where we represent more general graphs by labeled forests. The labels will always be integers or strings of integers chosen from a range which is linear in the size of the original graphs so the results of this section will apply. We will thus be able to obtain linear automorphism algorithms for a class of graphs larger than just forests.

3. Interval Graphs

An *interval graph* is a graph $G = (V,E)$ in which there is a one-to-one correspondence between the vertices V and some family of intervals on the real line. The correspondence must have the property that two vertices are adjacent in G if and only if their two corresponding intervals have a non-empty intersection. We will briefly review some facts about interval graphs here. Further background and additional references are given by Booth and Lueker [5,23]. They have shown how to reduce the problem of testing interval graph isomorphism to the problem of testing isomorphism between special labeled trees. Here we explain how to extend those methods to obtain linear algorithms for computing the automorphism partition, a set of generators for the automorphism group, and the order of the automorphism group.

The basis of their isomorphism algorithm is a procedure for representing an interval graph by a *PQ-tree.* The leaves of the *PQ*-tree are the *dominant* *(maximal) cliques* of the interval graph. Internal nodes are either *P-nodes* which, like nodes in normal trees, have an unordered set of children, or else they are *Q-nodes* which have three or more children whose left-to-right order is rigid up to a complete reversal. Figure 3.1 provides an example of an interval graph and a *PQ*-tree which represents it.

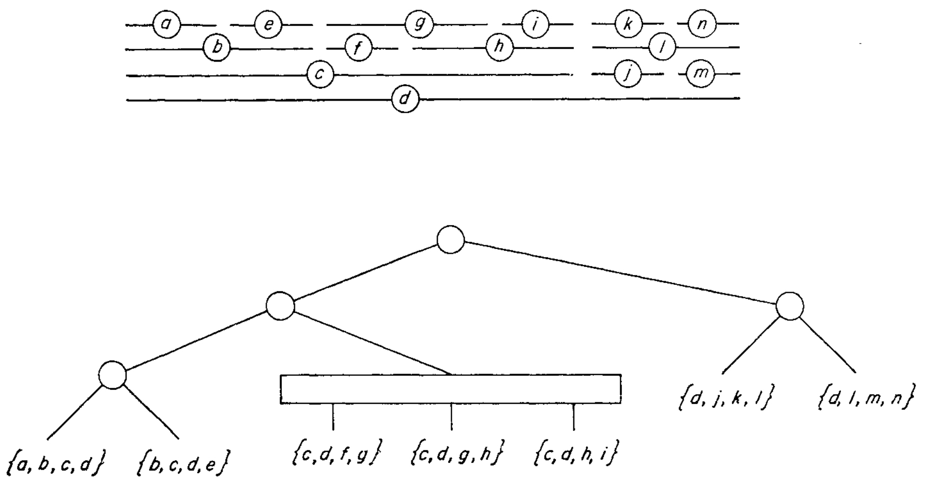


FIGURE 3.1: A set of intervals (above) and the *PQ*-tree (below) which represents the associated interval graph. Each of the seven leaves in the tree is a dominant clique from the interval graph. *P*-nodes are drawn as circles and *Q*-nodes are drawn as rectangles.

Throughout the remainder of this section we will assume that G is an interval graph. To avoid confusion we will depart somewhat from our previous notation and refer to the vertices of a *PQ*-tree as nodes so as to distinguish them from the vertices of G. We will further assume that all *PQ*-trees are drawn in the normal fashion with their roots at the top and their leaves at the bottom and that the leaves have a specific left-to-right order in this presentation. Every automorphism of a *PQ*-tree will thus correspond to an implicit re-drawing of the tree in which similar nodes occupy similar positions.

It has been shown [23] that the *PQ*-tree for an interval graph is unique up to isomorphism. An even stronger result can be proven which will enable us to solve the automorphism problems.

LEMMA: 3.1: Every automorphism of the *PQ*-tree for an interval graph G induces a distinct automorphism on G.

PROOF: Consider a particular embedding of the *PQ*-tree as discussed above. The leaves taken in left-to-right order form the *frontier* of the *PQ*-tree. Any automorphism of the *PQ*-tree will produce a different frontier and hence a different ordering of the dominant cliques.

Suppose that we have a particular frontier and that each clique is labeled with the sequence of degrees for its vertices, the degrees being specified in increasing order. We claim that the interval graph can be reconstructed from this information. The key observation is the fact, proven elsewhere [23], that the set of cliques to which each individual vertex belongs is always a consecutive set of leaves along the frontier. This is a basic property of the *PQ*-tree for an interval graph and forms the foundation for all of the *PQ*-tree algorithms.

```
procedure RECONSTRUCT( $\{C_i\}$, $k$ ):
  begin
    $n \leftarrow 0$;
    $E \leftarrow \emptyset$;
    $A \leftarrow \emptyset$;
    for $i \leftarrow 1$ until $k$ do
      begin
        for $j \in A$ do
          $C_i \leftarrow C_i - \{degree(j)\}$;
        while $C_i \neq \emptyset$ do
          begin
            $n \leftarrow n+1$;
            $degree(n) \leftarrow \min(C_i)$;
            $C_i \leftarrow C_i - \{degree(n)\}$;
            $edges(n) \leftarrow 0$;
            for $j \in A$ do
              begin
                $E \leftarrow E \cup \{\{j,n\}\}$;
                $edges(j) \leftarrow edges(j)+1$;
                $edges(n) \leftarrow edges(n)+1$
              end;
            $A \leftarrow A \cup \{n\}$;
            for $j \in A$ do
              if $edges(j)=degree(j)$ then $A \leftarrow A - \{j\}$
          end
      end
  end
```

FIGURE 3.2: A Pidgin Algol procedure [1] which reconstructs an interval graph from its *PQ*-tree. The first input is the sequence of degree sets for the cliques along the frontier of the *PQ*-tree. The second input is the number of cliques. The procedure produces the set of edges E for the graph and the number of vertices n. The vertices which are missing edges at any step are "active" and are placed in the set A. They continue to have edges added as new vertices are created until finally their number of edges is equal to their degree, at which point they become "inactive" and are removed from A.

Figure 3.2 contains a Pidgin Algol procedure which reconstructs an interval graph from the frontier of its *PQ*-tree. The reconstruction proceeds as follows. Starting with the leftmost clique a set of consecutively numbered vertices (beginning with one) is created for each degree of a vertex in the clique, the degrees being processing in increasing order. Edges are added between all created vertices to form a clique.

At a general step, having reached a new clique, the set of degrees is first modified by removing an instance of each degree for which there is a previously created vertex with the same original degree but which still has fewer edges than indicated by its degree. The requirement that vertices appear in consecutive cliques guarantees that there is no loss of generality in this step because all vertices which still have missing edges must be in this new clique. The process then proceeds as before, continuing the creation of consecutively numbered vertices in increasing order of degree and adding edges among all new vertices and also between all new vertices and all old vertices which are still missing edges.

An easy induction on the number of cliques verifies that the graph which is constructed is isomorphic to the original interval graph from which the *PQ*-tree was built. Every automorphism of the tree thus induces a permutation of the vertices which is an automorphism of the interval graph. Each of these interval graph automorphisms is distinct.

To be convinced of this, observe that every automorphism of the *PQ*-tree has a distinct frontier. If we consider two distinct automorphisms of the *PQ*-tree, at least one clique must be in a different position within the frontier. No two dominant cliques have identical sets of vertices. Thus the clique which differs in the two PQ-tree automorphism must have a vertex which is treated differently by the reconstruction procedure. It then follows that the two interval graph automorphisms differ on that vertex, implying that they are in fact distinct automorphisms. □

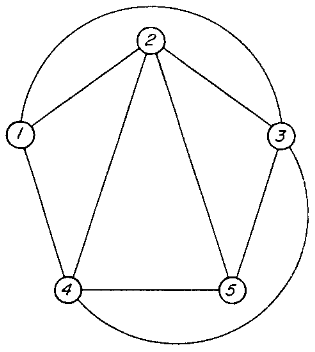


FIGURE 3.3: The interval graph produced by the reconstruction procedure of Figure 3.2 using the frontier of the *PQ*-tree from Figure 3.1. The procedure has been run for two iterations ($i \leftarrow 1, 2$) of the outermost for-loop. At this point in the execution $n = 5$, $A = \{3, 4\}$, and the degrees for vertices 1 through 5 are respectively 3, 4, 8, 13, and 3. If the algorithm is run to completion it will construct a graph having fourteen vertices. It is easy to check that this is an interval graph for which the intervals in Figure 3.1 form an *intersection model* which represents the graph [5].

Having discovered that every automorphism of the *PQ*-tree leads naturally to a different automorphism of *G* we might ask whether the converse is true. Does every automorphism of *G* induce a different automorphism of the *PQ*-tree? The answer is no. The reason can be traced to the way in which the interval graph was reconstructed from the tree. At various stages vertices were created to correspond to degrees of vertices within cliques. It can easily turn out that the same degree is used for more than one vertex being created for a particular clique. In the algorithm of Figure 3.2 this follows from the fact that the selection of the minimum of C_i is nondeterministic since C_i is in general a multiset of degrees. The reconstruction algorithm does not distinguish these vertices. They are similar but distinct, so some automorphism should interchange them. It is these additional automorphisms which we must characterize.

We need a bit more information concerning the *PQ*-tree for this task. Every vertex in *G* has a *characteristic node* in the *PQ*-tree. This is the unique node (there always is one) which roots the subtree whose leaves are exactly the cliques to which the vertex belongs [5,23]. Strictly speaking the term characteristic node is a bit imprecise because some vertices have characteristic nodes which are actually only part of a *Q*-node. The rigid left-to-right order of a *Q*-node's children means that only some of them (always a consecutive subset) have leaves which contain the vertex in question. Nevertheless we will use the term characteristic node to mean the leaf, *P*-node, or portion of a *Q*-node which contains those cliques. Figure 3.4 illustrates the characteristic nodes for the interval graph shown earlier.

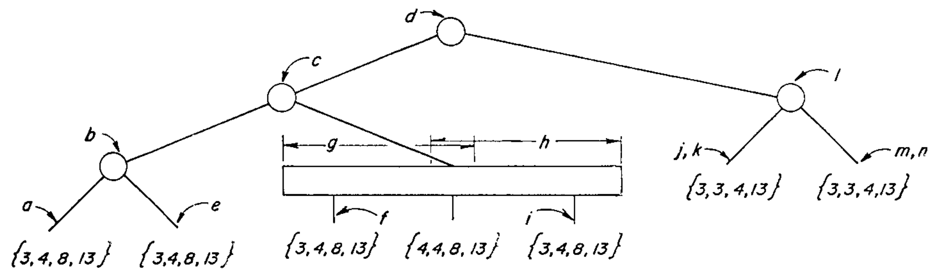


FIGURE 3.4: The characteristic nodes for the interval graph in Figure 3.1. Note that intervals *g* and *h* have characteristic nodes which are portions of the *Q*-node. The characteristic node for *g* is the two leftmost children of the *Q*-node whereas the characteristic node for *h* is the two rightmost children.

LEMMA: 3.2: Every automorphism of *G* is completely determined by an automorphism of the *PQ*-tree for *G* together with a permutation of the vertices which preserves characteristic nodes.

PROOF: Consider an automorphism of the interval graph *G*. It can be decomposed into two automorphisms, one on the *PQ*-tree and one which preserves characteristic nodes, in the following manner.

The automorphism induces an automorphism on the *PQ*-tree. This in turn induces an automophism of G, which is not necessarily the same as the originial automorphism because because the reconstruction has lost some information distinguishing vertices belonging to the same set of cliques. Vertices which are not distinguished by the reconstruction procedure are precisely those vertices which have identical characteristic nodes. They are always similar. Moreover, they may be arbitrarily permuted within the cliques with no change to the reconstruction. Thus the particular automorphism can be obtained from the automorphism induced by the reconstruction by a reorderingof each set of vertices having the same characteristic node.

It follows immediately from these remarks that the additional shuffling of these vertices accounts for all possible automorphisms of G. □

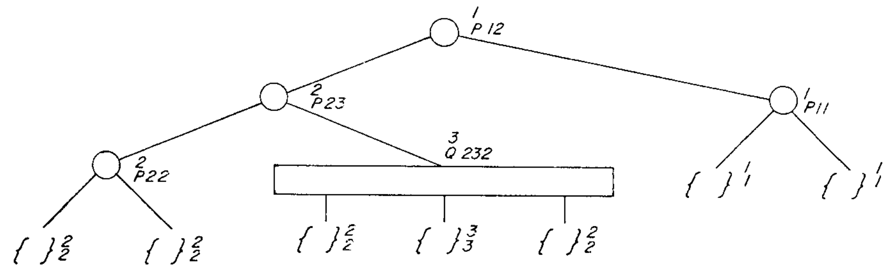


FIGURE 3.5: The i-numbering for the PQ-tree of Figure 3.1. Internal nodes have working labels which begin with a P or a Q to differentiate the two types of nodes. The remainder of a P-node's working label consists of the i-numbers, in increasing order, of the children. The i-numbers in the working label of a Q-node are in left-to-right order of the children and thus not always in increasing order.

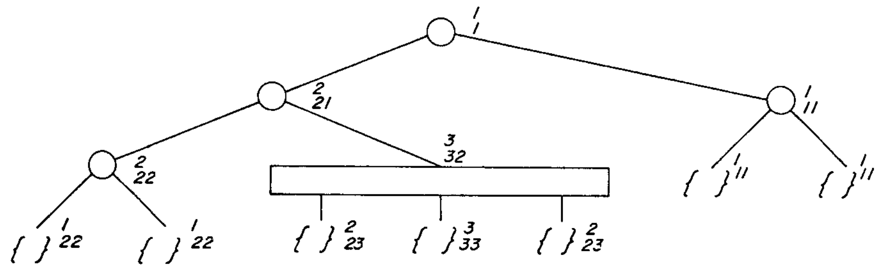
We are now ready to state the algorithm for computing the automorphism partition of an interval graph. We first label each leaf with the degrees of its vertices in increasing order. The i-numbering then proceeds as before followed by j-numbering. Two vertices are similar if and only if their characteristic nodes are similar in the PQ-tree. The j-numbers and depth information give this information.

LEMMA: 3.3: The automorphism partition of an interval graph can be found in linear time.

PROOF: Booth and Lueker prove that the PQ-tree of an interval graph can be constructed in linear time and that the characteristic nodes can be found for each vertex within the same time bound [5,23]. The rest of the calculation is linear since it is a variation on the computation of the automorphism partition of a labeled tree.

Working labels for P-nodes consist of the letter P followed by the i-numbers of the children in increasing order. Working labels for Q-nodes consist of the letter Q followed by the i-numbers of the children in left-to-right or right-to-left order, whichever gives the lexicographically smaller label. The j-numbering is identical to the earlier calculation for trees.

Vertices of the interval graph are partitioned by assigning each one a label which consists of the similarity class for the characteristic node. If the characteristic node is a *P*-node this is trivial. If it is a *Q*-node the label must distinguish the portion of the *Q*-node (a consecutive set of children) comprising the characteristic node. This is accomplished by numbering the children from left-to-right (or right-to-left, depending upon how the working label was generated during the *i*-numbering) and appending the index of the left-most and right-most children to the label for the vertex. A final bucket sort accomplishes the partitioning into similarity classes. □



The blocks of the automorphism partition of the interval graph

$\{a,e\}$ $\{b\}$ $\{c\}$ $\{d\}$ $\{f,i\}$ $\{g,h\}$ $\{j,k,m,n\}$ $\{l\}$

FIGURE 3.6: The *j*-numbering for the *PQ*-tree of Figure 3.1. The eight blocks of the automorphism partition for the original interval graph are shown below the *PQ*-tree.

Finding a set of generators for the automorphism group is an easy generalization of the automorphism algorithm for a forest. There are two minor variations to the basic forest algorithm which must be made. The first is that a set of generators must be added which permute each family of vertices having the same characteristic node. A bucket sort produces the families and the compact representation for a set of generators is then easily produced. This can be done before the *i*-numbering so we will not worry about this contribution to the complexity since it is obviously linear.

The second variation is that as the generators are produced there can arise a situation which does not occur in the normal algorithm. Some *Q*-nodes can have their children reversed left-to-right and still have the resulting subtree isomorphic to the original subtree. When this happens it is necessary to output the compact representation of an automorphism which performs this reversal. This is easy to do by walking the tree but we need to verify that the algorithm is still linear. The earlier argument that each vertex is in the range of an automorphism at most once no longer holds. But careful analysis shows that each time a *Q*-node is reversible we are guaranteed that one half of its children will disappear at the next level because they must be similar to the other half of the children. If there are an odd number of children the middle child does not have to participate in the reversal because it maps to itself. This is enough to ensure linearity because each vertex is now in the range at most twice.

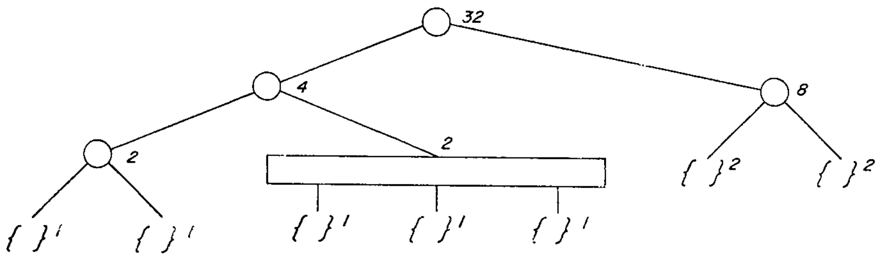


FIGURE 3.7: The k-numbering of the PQ-tree of Figure 3.1. The order of the automorphism group of the interval graph is the k-number of the root. In this example there are 32 distinct automorphisms.

Counting the number of automorphisms for an interval graph is equally straightforward. A k-numbering of the PQ-tree determines the order of its automorphism group and the remaining automorphisms can be counted by multiplying by the total number of permutations which preserve characteristic nodes. This is simply the product of the factorials of all families of vertices having a common characteristic node and is again a linear time computation, modulo the entire discussion of Section 2 regarding the computation of factorial. In practice it is simpler to include this in the k-numbering. The k-number of each node is multiplied by the factorial of the number of vertices for which it is a characteristic node. Q-nodes will have a slightly more complicated k-number because they can correspond to more than a single characteristic node in which case the product of the factorials is used. This modified k-numbering is illustrated in Figure 3.7.

These PQ-tree algorithms can be extended to handle labeled interval graphs in an obvious manner. Instead of each clique receiving a label which consists only of degrees we first bucket sort all of the vertices' labels and assign ranks which are then added to the degree information labeling the cliques. The remainder of the algorithm is the same except that two vertices are similar if and only if they have the same label and similar characteristic nodes. We have proven the following.

THEOREM: 3.4: It is possible to test isomorphism, to find the automorphism partition or a set of generators for the automorphism group, and to count the number of automorphisms or isomorphisms for labeled interval graphs in time and space which is linear in the size of the interval graphs plus the sum of the lengths of the labels.

4. Outerplanar Graphs

An *outerplanar graph* is a graph having a planar embedding in which all of the vertices lie on the exterior face [11]. These graphs are obviously planar so we know (jumping ahead to Section 5) that they have linear time automorphism algorithms. But algorithms to handle the general case are fairly complicated and any implementation is likely to be quite intricate. However, outerplanar graphs are a greatly restricted subset of the planar graphs and we can take advantage of their additional structure to produce algorithms which are simpler than the corresponding algorithms for labeled planar graphs.

Our outerplanar algorithms have the same linear asymptotic running time as do the planar algorithms but the constants of proportionality are much lower compared with the more general algorithms. In this respect we offer quite practical solutions to the isomorphism and automorphism problems for outerplanar graphs. These new algorithms, like much of the previous work on outerplanar graphs, are based on a result of Tang [32].

LEMMA: 4.1: If G is a graph in which there are at most two vertex-disjoint paths of length greater than one between each pair of vertices then

(1) G is planar,

(2) $e \leqslant 2n - 2$, and

(3) if G is biconnected and $n \geqslant 5$ then G has a unique Hamilton cycle.

It is easy to see that outerplanar graphs satisfy the hypotheses of the lemma. Any two vertex-disjoint paths must lie on different portions of the exterior face and thus there are at most two such paths joining any pair of vertices. We can state the following result.

COROLLARY: 4.2: Every biconnected outerplanar graph having at least three vertices possesses a unique Hamilton cycle.

PROOF: For five or more vertices Tang's lemma guarantees that there is a unique Hamilton cycle. The remaining cases are verified exhaustively. The only biconnected outerplanar graphs having at least three but no more than four vertices are the graphs K_3, $K_{2,2}$, and K_4-x. Each of these has a unique Hamilton cycle as shown in Figure 4.1. □

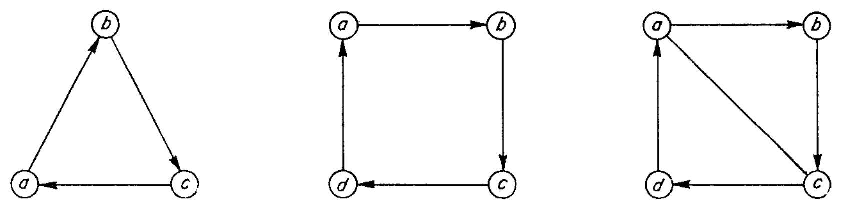


FIGURE 4.1: The three biconnected outerplanar graphs on three or four vertices with their unique Hamilton cycles. Edges within the cycles are indicated by the directed arrows.

For an arbitrary outerplanar graph standard depth-first search algorithms will produce the biconnected components in linear time [1,33]. Using the unique Hamilton cycles we will show that each of these biconnected components can be placed into a canonical form from which information about the automorphisms can be easily extracted. These in turn can be used to build a canonical tree representation for each of the connected components so that the resulting forest is a canonical representation for the entire graph. From this forest, automorphism information can be easily obtained using algorithms similar to those developed in Sections 2 and 3. We thus examine the problem of placing the biconnected components into canonical form, putting off until later the task of piecing this information back together again for the whole graph.

Given a biconnected outerplanar graph the unique Hamilton cycle can be found in linear time by successively removing vertices of degree two. Beyer, Jones, and Mitchell use this approach to test isomorphism of maximal outerplanar graphs [2]. They rely upon the fact that the sequence of vertex degrees encountered along the Hamilton cycle, the *Hamilton degree sequence*, completely characterizes the graph. Unfortunately this characterization does not extend to the more general case of biconnected outerplanar graphs. Figure 4.2 illustrates this situation, showing two non-isomorphic biconnected outerplanar graphs which have the same Hamilton degree sequence.

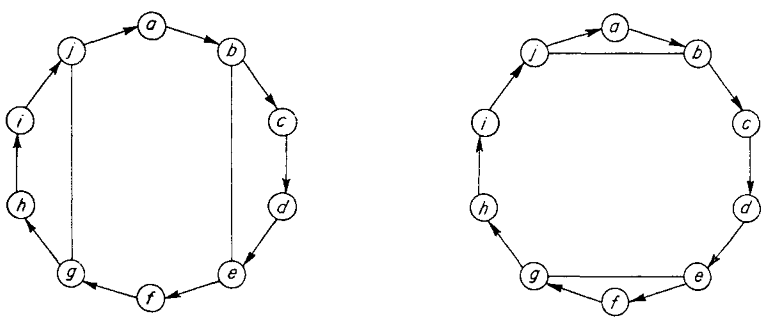


FIGURE 4.2: Two non-isomorphic biconnected outerplanar graphs having the same Hamilton degree sequence 2322323223.

Syslo [31] gave a similar example in his description of a linear time coding algorithm (and hence a linear time isomorphism test) for biconnected outerplanar graphs. He again used the unique Hamilton cycle, but he proposed a more elaborate scheme which uses additional information concerning the structure of the graph. After identifying the unique Hamilton cycle and numbering the vertices accordingly his algorithm performs a series of reduction and labeling steps to finally obtain a string representation which is then minimized and used as a canonical representation for the graph.

We think that our approach is somewhat simpler and leads more directly to efficient automorphism algorithms. We observe that although the Hamilton degree sequence is not sufficient to completely determine a biconnected outerplanar graph we certainly can reconstruct the graph if we have a list of the adjacencies for each vertex along the cycle. The problem of course is to represent the adjacencies in a manner which is independent of the exact starting position chosen for the cycle since otherwise there might be as many as n different representations.

To this end we represent the adjacencies at each vertex by a list which consists of the distances along the Hamilton cycle between the vertex and each of its neighbors. We further stipulate that these distances be in increasing order. Notice that the lists depend upon the orientation chosen for the Hamilton cycle but not upon its starting point. Figure 4.3 shows the adjacency lists for the two graphs of Figure 4.2. Each vertex is assigned two lists, the first for a clockwise traversal of the Hamilton cycle and the second for a counter-clockwise traversal.

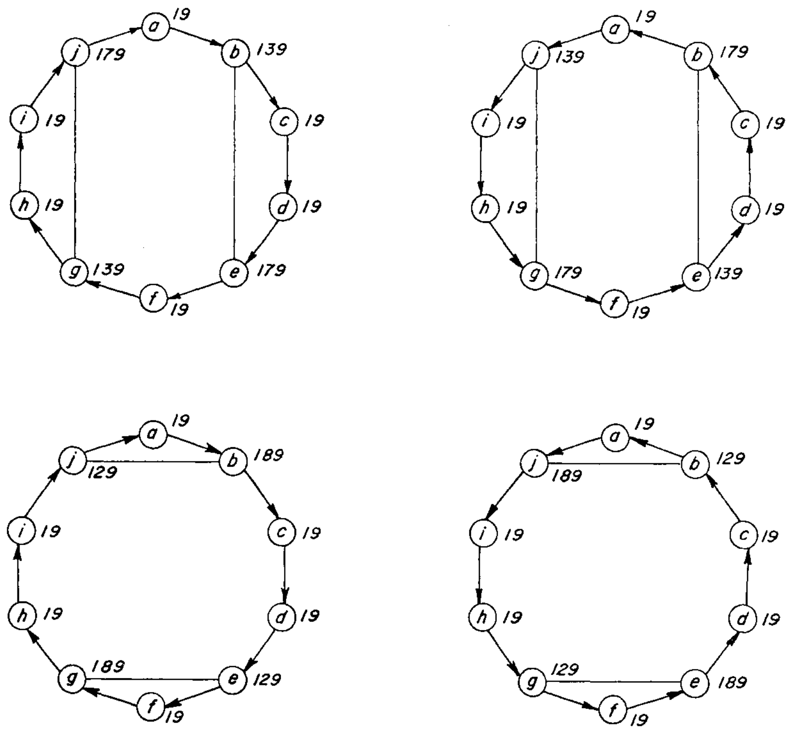


FIGURE 4.3: Clockwise (on the left) and counter-clockwise (on the right) Hamilton adjacency lists for the two graphs of Figure 4.2. The fact that the two sequences are different implies that the top graph is not isomorphic with the bottom graph.

In general we will let S be the sequence obtained by following the Hamilton cycle in one direction, concatenating adjacency lists, and we will let R be the sequence obtained by concatenating in the opposite direction, using the reversed adjacency lists. These are the *Hamilton adjacency sequences* for the graph. Isomorphism of biconnected outerplanar graphs can be characterized directly in terms of these sequences.

LEMMA: 4.3: Let G_1 and G_2 be biconnected outerplanar graphs and suppose that S_1 and S_2 are the Hamilton adjacency sequences for their unique Hamilton cycles and that R_1 and R_2 are the Hamilton adjacency sequences for the reversed Hamilton cycles. Then G_1 is isomorphic to G_2 if and only if S_1 is either a cyclic shift of S_2 or a cyclic shift of R_2.

PROOF:

Clearly two isomorphic graphs must have Hamilton adjacency sequences which are either cyclic shifts or reversals of each other. But as we have already remarked, given the Hamilton adjacency sequence of a graph, the graph can be reconstructed up to isomorphism. □

The unique Hamilton cycle in a biconnected outerplanar graph can be found in linear time by successively removing vertices of degree two, being careful to keep track of vertices which must be adjacent in the cycle [2,27,31]. The sequences S_1, S_2, R_1, and R_2 can be constructed at the same time by numbering the vertices along the cycle and bucket sorting the distances for each adjacency, building the lists for each of the vertices by adding the smallest distances first.

Restating the last lemma as a pattern matching problem we see that two graphs are isomorphic if and only if S_1 is a substring of $S_2S_2\$R_2R_2$. This condition can be tested using any one of many algorithms for linear time pattern matching [1, Chapter 9]. Noting that all of the objects are linear in the size of the graphs (the strings S and R each have length $2e$) and that the graphs themselves are linear in n (by Lemma 4.1) we have proven the following result.

COROLLARY: 4.4: Two biconnected outerplanar graphs can be tested for isomorphism in $O(n)$ steps.

Syslo's coding algorithm achieves the same asymptotic running time but we claim that our algorithm is easier to implement and that it will have a smaller multiplicative constant. But what is more important our algorithm is easily extended to solve the automorphism problems discussed in earlier sections. All that is required is a little more information from the pattern matching algorithms. Let p_i be the position within S in which the adjacency list for vertex i begins and let q_i be the position in R in which the reversed adjacency list for vertex i begins.

LEMMA: 4.5: Let G be a biconnected outerplanar graph, let S be its Hamilton adjacency sequence, and let R be its reversed Hamilton adjacency sequence. Two vertices i and j are similar if and only if the substring of length $2e$ which begins in position p_i of SS is identical with the substring of length $2e$ which begins either in position p_j of SS or in position q_j of RR.

One way to compute the similarity classes is to find the position tree for the string $SS\$RR$ using Weiner's algorithm [1,37]. Two vertices are in the same similarity class whenever they have a common ancestor at depth $2e$ in the position tree. The standard algorithm for finding position trees requires time proportional both to the length of the input string and to the alphabet size. Since our strings have symbols ranging from 1 to n this would imply an $\Omega(n^2)$ running time.

But this is more machinery than we need bring to bear on this problem. Any automorphism must map a Hamilton cycle onto a Hamilton cycle. Thus for biconnected outerplanar graphs the only possible automorphisms are cyclic shifts and reversals of the Hamilton cycle. It is sufficient to compute just the similarity class of vertex 1; the class for vertex j is found by adding $j-1$ (modulo n) to all of the vertices in the first class if the automorphism corresponds to shifting the Hamilton cycle; if the automorphism corresponds to reversing the Hamilton cycle the vertices in each class must be "reversed" in the obvious manner.

It follows that the automorphism group is always a subgroup of a dihedral group, the $2n$ permutations which cyclically shift and/or reverse a cycle of length n. is The number of automorphisms for a biconnected outerplanar graph is the number of distinct cyclic shifts and reversals which map the Hamilton cycle onto itself. We can find the order of the group by counting the number of times that S occurs as a substring in $SS'\$RR'$ where S' and R' are the respective strings S and R with their final symbols removed to avoid counting the identity and reversal automorphisms twice.

Guy [10] gave similar counting results for automorphisms of maximal outerplanar graphs, showing that there were at most six automorphisms. This is a very special case because the more general biconnected outerplanar graphs can realize the full dihedral group, as evidenced by the n-cycle which is biconnected, outerplanar, and has $2n$ automorphisms.

For the automorphism groups of biconnected outerplanar graphs a set of generators consists of a single cyclic shift of the minimum distance and (if appropriate) a reversal of the cycle. Again, these two automorphisms are easily found using linear time pattern matching.

LEMMA: 4.6: There exist linear time algorithms to compute the automorphism partition, to determine the order of the automorphism group, and to find a set of generators for the automorphism group of a biconnected outerplanar graph.

We turn our attention next to the problem originally solved by Syslo, that of producing a coding for biconnected outerplanar graphs in linear time. The Hamilton adjacency sequence is almost a canonical form for a biconnected outerplanar graph. If we encode the graph by finding the starting vertex and direction for the Hamilton cycle which produces the lexicographically smallest adjacency sequence we will produce a coding. It is easy to modify the Knuth-Morris-Pratt pattern matching algorithm to have it find the lexicographically smallest cyclic shift of a string in linear time and space; there are other algorithms, the best known to us is due to Shiloach [4,28,29]. This yields a linear time coding algorithm for biconnected outerplanar graphs.

We conclude this section with a sketch of the automorphism algorithms for general outerplanar graphs. The biconnected components can be used to build a tree (a variant of the *block cut-vertex tree)* for each connected component. These trees form a forest representing the entire graph.

Biconnected components are labeled with their Hamilton adjacency sequence which has been shifted so it begins with the unique cut vertex which is the parent in the tree. The Hamilton adjacency sequence is reversed if necessary to give the smallest possible lexicographic value. The exception to this rule is a biconnected component which has no parent in its tree (there is at most one in each tree). These are represented by the shift or reversal of the sequence which gives a lexicographically least value after the i-numbers for the children have been inserted

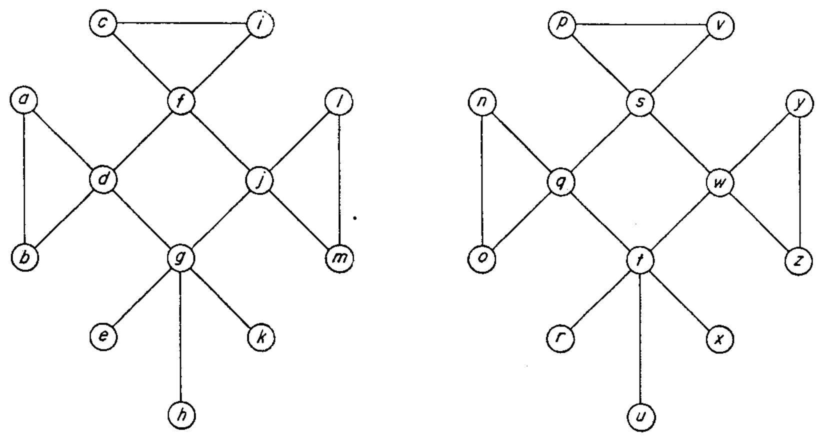


FIGURE 4.4: A general outerplanar graph having two connected components and fourteen biconnected components.

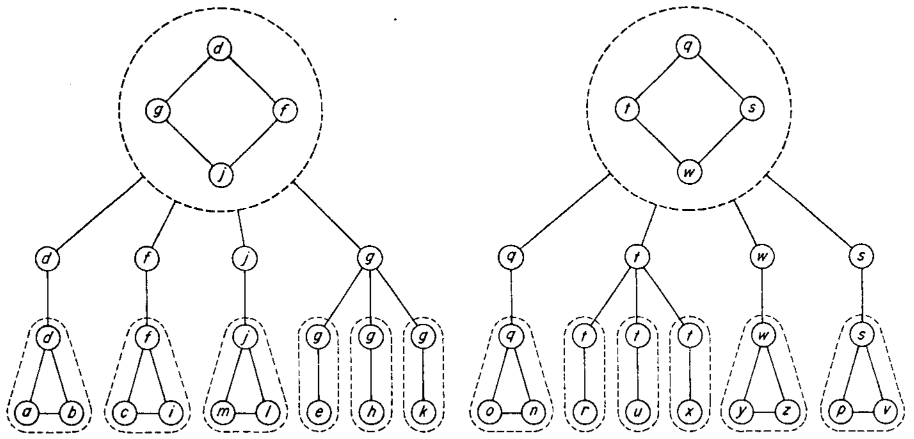


FIGURE 4.5: The block cut-vertex forest for the outerplanar graph of Figure 4.4. Each node within the forest is either a block of the graph or a cut vertex of the graph. Blocks (biconnected components) are indicated by dotted lines.

into the Hamilton adjacency sequence.

An isomorphism test is a straighforward modification of the earlier labeled tree isomorphism algorithm. At each level in the forest, beginning at the bottom,

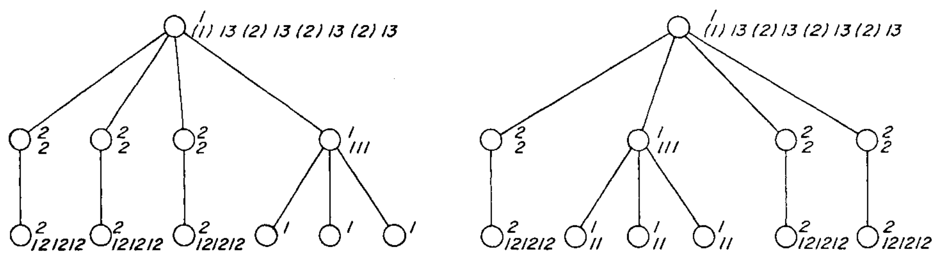


FIGURE 4.6: The *i*-numbering for the block cut-vertex forest of Figure 4.5. The code for this forest is a canonical representation of the original graph. The working label of a block is its Hamilton adjacency sequence. For non-root blocks, the sequence begins with the unique cut vertex which is the parent of the block; a root block is labeled with its lexicographically least sequence. In non-leaf blocks the adjacency list of a cut vertex is preceded by the *i*-number of the cut vertex, contained within parentheses. Working labels for cut vertices consist of the *i*-numbers of the children in increasing order.

the labels are bucket sorted and assigned ranks. A check is made that the multiset of labels is identical for the two forests. The *i*-numbers are then passed up the tree, inserted appropriately into the parents' labels (being careful to distinguish ranks from distances) and the next level is bucket sorted. This continues until the roots are the only unprocessed nodes. At the top level the labels with rank information inserted are each placed into a canonical form (minimum lexicographic order) and then bucket sorted. Two outerplanar graphs are isomorphic if and only if their forests generate the same multiset of labels for the roots.

Algorithms for computing the automorphism partition, counting the number of automorphisms, and producing a set of generators for the automorphism group are easy to build using the forests and the automorphism algorithms from Section 2. Labeled outerplanar graphs are handled by incorporating the ranks of the bucket sorted labels into the Hamilton adjacency sequences in an obvious manner. The modifications are quite similar to those used for interval graphs. We can state the following result.

THEOREM: 4.7: It is possible to test isomorphism, to find the automorphism partition or a set of generators for the automorphism group, and to count the number of automorphisms or isomorphisms for labeled outerplanar graphs in time and space which is linear in the size of the outerplanar graphs plus the sum of the lengths of the labels.

5. Planar Graphs

In this section we will briefly sketch the techniques which might be used to build linear time automorphism algorithms for general planar graphs. The algorithms we propose are admittedly quite complicated. We have not implemented them; to do so will require that many details be worked out which we have not fully thought out. Nevertheless, we maintain that the basic ideas sketched here form a convincing argument that the automorphism problems for the general case of planar graphs can also be solved in linear time.

Our methods build upon those developed by Hopcroft, Tarjan, and Wong [16,20] which were used to produce linear time algorithms for testing isomorphism of planar graphs. Fontet [9] has also used their results as the basis of a linear algorithm for finding the automorphism partition of a planar graph. His method is quite similar to ours, but he bases his construction on the Hopcroft-Tarjan algorithm whereas we favor the Hopcroft-Wong solution. We indicate algorithms for solving all of the automorphism problems.

To review, here is a short survey of the development of planar graph isomorphism algorithms. In 1970 Hopcroft [13] demonstrated that the states of a finite automaton could be minimized in $O(n \log n)$ time. The application of this technique to planar graph isomorphism was pursued in a series of papers by Hopcroft and Tarjan [14-17]. They reduced the isomorphism problem for planar graphs to the problem for triconnected planar graphs by using a tree representation in which each leaf corresponds to a maximal triconnected subgraph. The connected, biconnected, and triconnected components are all found in linear time [18,33]. The problem was further reduced to isomorphism of triconnected planar embeddings using Whitney's theorem that a triconnected planar graph has a unique embedding [19,38].

Hopcroft and Tarjan were only able to show that planar graphs can be tested for isomorphism in $O(n \log n)$ time [17] but Hopcroft and Wong were able to obtain linear algorithms using the planar embeddings [20,39]. The Hopcroft-Tarjan algorithm produces the automorphism partition of the triconnected embedding as part of the isomorphism test. Fontet [9] later improved the time bound for Hopcroft and Tarjan's method to $O(n)$ by establishing some interesting properties of planar embeddings. He was the first to show that the automorphism partition of a planar graph can be found in linear time.

We take a different approach than Fontet and base our automorphism algorithms on the Hopcroft-Wong algorithm. Necessarily we will only be able to sketch our procedure. A more complete description and a proof of correctness would require a more thorough analysis of the Hopcroft-Wong algorithm than has yet appeared in the literature. Our algorithms will use the linear time forest automorphism algorithms along with linear pattern matching.

The overall algorithm follows Hopcroft and Wong. The connected, biconnected, and triconnected components are found and used to construct a forest, similar to the forest used for outerplanar graphs, which is then i-numbered, j-numbered, and k-numbered. The difficult part is the way in which the triconnected planar embeddings are handled.

Hopcroft and Wong's algorithm successively reduces the planar embeddings to smaller and smaller embeddings by applying two operations which remove loops or multiple edges and vertices of low degree. The information from these reductions is kept so that the graph can be reconstructed. A second critical step is the application of a *circle isomorphism* procedure which determines whether two labeled cycles are isomorphic. The overall algorithm eventually tests the entire graph by winding back through the reductions, propagating information obtained at lower levels in the algorithm.

Wong was able to show that every triconnected planar embedding must either have an applicable reduction or else it must be one of three simple graphs: a labeled vertex, a labeled cycle, or a Platonic solid [39]. This guarantees that the test is linear since all of these cases, with the exception of the labeled cycles, involve only graphs of bounded size.

We propose using linear pattern matching to place all of the circles into a canonical form by finding their lexicographically smallest shift and using that shift to represent the cycle. Isomorphism and automorphism questions for circles are then easily answered as discussed in Section 4. The result is a canonical *i*-numbering of the tree which represents the planar graph. After *j*-numbering and *k*-numbering we can easily find all of the other automorphism information for the graph. We state without proof our main result concerning planar graphs.

PROPOSITION: 5.1: It is possible to test isomorphism, to find the automorphism partition or a set of generators for the automorphism group, and to count the number of automorphisms or isomorphisms for labeled planar graphs in time and space which is linear in the size of the planar graphs plus the sum of the lengths of the labels.

This section has been only a sketch of the methods we propose. The details are similar to those given in previous sections but are significantly more complicated due to the extra complexity of finding triconnected components and other complexities inherent in the Hopcroft-Wong algorithm. We claim that our version of circle isomorphism may be an improvement over the original version and that our algorithms in principle can be used to solve all of the automorphism problems, in contrast with Fontet's solution which finds only the automorphism partition. A verification of these claims awaits further work to clarify the many details omitted here.

6. Concluding Remarks

We have shown that four classes of graphs having linear time isomorphism tests have linear time algorithms to compute the automorphism partition, to count the number of automorphisms, to produce a set of generators for the automorphism group, and also to determine codings for the graphs. The algorithms for outerplanar graphs, particularly for the biconnected case, are very easy to implement and should be much faster than the algorithms for the general planar case.

We conclude with a few remarks suggesting possible areas for future investigation. Polya [26] showed that the automorphism group of a tree can always be expressed as the direct or wreath products of symmetric groups. Weinberg, Harary, and Tutte [12,36] showed that triconnected planar graph have automorphism groups which are linear in size. Biconnected outerplanar graphs have automorphism groups which are always subgroups of a dihedral group and maximal outerplanar graphs have at most six automorphisms [10].

These examples suggest that there is a relationship between the "complexity" of the automorphism group and the "complexity" of testing for isomorphism. In particular we conjecture that for classes of graphs in which the isomorphism problem is equivalent to the general graph isomorphism problem it will turn out that in some vague sense almost every group will be the automorphism group of some graph within the class. A precise statement of this proposition awaits further research.

7. Acknowledgements

We would like to express our appreciation to Marlene Colbourn, Ian Munro, Brian Plante, and Ron Read for assistance with various aspects of this research. Leo Guibas first told us of the work of Shiloach.

References

[1] A. V. AHO, J. E. HOPCROFT, and J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.

[2] T. BEYER, W. JONES, and S. MITCHELL, A linear time algorithm for isomorphisms of maximal outerplanar graphs, *Journal of the ACM*, 26:4, pp. 603-610, October 1979.

[3] J. A. BONDY and U. S. R. MURTY, *Graph Theory With Applications*, MacMillan, London, 1976.

[4] K. S. BOOTH, Finding a lexicographic least shift of a string, submitted for publication.

[5] K. S. BOOTH and G. S. LUEKER, Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms, *Journal of Computer and System Sciences* 13:3, pp. 335-379, December 1976.

[6] G. BUSACKER and T. L. SAATY, *Finite Graphs and Networks*, McGraw Hill, 1965.

[7] C. J. COLBOURN, Graph generation, M.Math. Thesis, University of Waterloo, 1977.

[8] D. G. CORNEIL, *Graph Isomorphism*, Ph.D. Thesis, University of Toronto, 1968.

[9] M. FONTET, Linear algorithms for testing isomorphism of planar graphs, *Proceedings Third Colloquium on Automata, Languages, and Programming*, pp. 411-423, 1976.

[10] R. GUY, Dissecting a polygon into triangles, *Bulletin of Malayan Mathematics Society* 5, pp. 56-60, 1958.

[11] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, Massachusetts, 1969.

[12] F. HARARY and W. T. TUTTE, On the order of the group of a planar map, *Journal of Combinatorial Theory*, 1:3, pp. 394-395, November 1966.

[13] J. E. HOPCROFT, An $n \log n$ algorithm for minimizing states in a finite automaton, in Z. KOHAVI and A. PAZ, editors, *Theory of Machines and Computations*, Academic Press, 1971.

[14] J. E. HOPCROFT, An $n \log n$ algorithm for isomorphism of planar triply connected graphs, Computer Science Technical Report STAN-CS-71-192, Stanford University, 1971.

[15] J. E. HOPCROFT and R. E. TARJAN, A V^2 algorithm for determining isomorphism of planar graphs, *Information Processing Letters* 1:1, pp. 32-34, February 1971.

[16] J. E. HOPCROFT and R. E. TARJAN, Isomorphism of planar graphs, in R. E. MILLER and J. W. THATCHER, editors, *Complexity of Computer Computations*, Plenum Press, pp. 131-151, 1972.

[17] J. E. HOPCROFT and R. E. TARJAN, A $V \log V$ algorithm for isomorphism of triconnected planar graphs, *Journal of Computer and System Sciences* 7:3, pp. 323-331, June 1971.

[18] J. E. HOPCROFT and R. E. TARJAN, Dividing a graph into triconnected components, *SIAM Journal on Computing* 2:3, pp. 135-158, September 1973.

[19] J. E. HOPCROFT and R. E. TARJAN, Efficient planarity testing, *Journal of the ACM* 21:4, pp. 549-568, October 1974.

[20] J. E. HOPCROFT and J. K. WONG, Linear time algorithm for isomorphism of planar graphs, *Proceedings Sixth Annual ACM Symposium on Theory of Computing*, pp. 172-184, 1974.

[21] K. R. JAMES and W. RIHA, Algorithm for description and ordering of trees, Technical Report 54, University of Leeds, 1974.

[22] J. LEDERBERG, Notational algorithms for tree structures, NASA Technical Report CR57029, 1964.

[23] G. S. LUEKER and K. S. BOOTH, A linear time algorithm for deciding interval graph isomorphism, *Journal of the ACM*, 26:2, pp. 183-195, April 1979.

[24] R. A. MATHON, A note on the graph isomorphism counting problem, *Information Processing Letters* 8:3, pp. 131-132, March 1979.

[25] S. MITCHELL, *Algorithms on Trees and Maximal Outerplanar Graphs*, Ph.D. Thesis, University of Virginia, 1977.

[26] G. POLYA, Kombinatorische Anzahlbestimmungen für Gruppen, Graphen, und Chemishe Verbindungen, *Acta Mathematica* 68, pp. 154-245, 1937.

[27] R. C. READ and D. G. CORNEIL, The graph isomorphism disease, *Journal of Graph Theory* 1:4, pp. 339-363, Winter 1977.

[28] Y. SHILOACH, A fast equivalence-checking algorithm for circular lists, *Information Processing Letters* 8:5, pp. 236-238, June 1979.

[29] Y. SHILOACH, Fast canonization of circular strings, submitted for publication.

[30] H. I. SCOINS, Placing trees in lexicographic order, in D. MICHIE, editor, *Machine Intelligence* 3, pp. 43-60, 1968.

[31] M. M. SYSLO, Linear time algorithm for coding outerplanar graphs, *Proceedings International Colloquium on Graph Theory Applications*, Ilmenau, 1977.

[32] D. T. TANG, Bi-path networks and multicommodity flows, *IEEE Transactions on Circuit Theory* 11, pp. 468-474, 1964. Our Lemma 4.1 is also given as Problem 11.26 by Harary [11].

[33] R. E. TARJAN, Depth-first search and linear graph algorithms, *SIAM Journal on Computing* 1:2, pp. 146-160, June 1972.

[34] L. WEINBERG, Algorithms for determining automorphism groups of planar graphs, *Proceedings of Third Annual Allerton Conference on Circuit and System Theory*, pp. 913-929, 1965.

[35] L. WEINBERG, Alternative algorithms for determining isomorphisms and automorphism groups of planar graphs, *Proceedings of Fourth Annual Allerton Conference on Circuit and System Theory*, pp. 444-453, 1966.

[36] L. WEINBERG, On the maximum order of the automorphism group of a planar triply-connected graph, *SIAM Journal on Applied Mathematics* 14:4, pp. 729-738, July 1966.

[37] P. WEINER, Linear pattern matching algorithms, *Proceedings Fourteenth Annual Symposium on Switching and Automata Theory*, pp. 1-11, 1973.

[38] H. WHITNEY, A set of topological invariants for graphs, *American Journal of Mathematics* 55, pp. 231-235, 1933.

[39] J. K. WONG, *Isomorphism Problems Involving Planar Graphs*, Ph.D. Thesis, Cornell University, 1975.