A Linear Time Algorithm For Finding Maximal Planar Subgraphs

Wen-Lian Hsu[1]

Institute of Information Science, Academia Sinica, Taipei, Taiwan, ROC
email: hsu@iis.sinica.edu.tw

ABSTRACT. Given an undirected graph G, the maximal planar subgraph problem is to determine a planar subgraph H of G such that no edge of G-H can be added to H without destroying planarity. Polynomial algorithms have been obtained by Jakayumar, Thulasiraman and Swamy [6] and Wu [9]. O(*mlogn*) algorithms were previously given by Di Battista and Tamassia [3] and Cai, Han and Tarjan [2]. A recent O($m\alpha(n)$) algorithm was obtained by La Poute [7]. Our algorithm is based on a simple planarity test [5] developed by the author, which is a vertex addition algorithm based on a depth-first-search ordering. The planarity test [5] uses no complicated data structure and is conceptually simpler than Hopcroft and Tarjan's path addition and Lempel, Even and Cederbaum's vertex addition approaches.[1]

1. Introduction

Given an undirected graph, the ***planarity testing problem*** is to determine whether there exists a clockwise edge ordering around each vertex such that the graph can be drawn in the plane without any crossing edges. Linear time planarity testing algorithm was first established by Hopcroft and Tarjan [4] based on a "***path addition*** approach". A "***vertex addition*** approach", originally developed by Lempel, Even and Cederbaum [8], was later improved by Booth and Lueker [1] to run in linear time using a data structure called a "PQ-tree". These approaches are quite involved. We developed a very simple linear time algorithm [5] based only on a depth-first search tree. The key to our approach is to add vertices according to a postordering obtained from a depth-first-search tree. Given an undirected graph G, the ***maximal planar subgraph problem*** is to determine a planar subgraph H of G such that no edge of G-H can be added to H without destroying planarity. Polynomial time algorithms have previously been given by [2,3,6,9] and a recent O($m\alpha(n)$) algorithm was obtained by La Porte [7]. We solved the MPS problem in linear time by modifying the algorithm in [5]. In the next section we briefly describe the idea in [5]. Our MPS algorithm is discussed in Sections 3 and 4. Finally, the time complexity is discussed in Section 5.

2. A Review of Our Modified Vertex Addition Approach for Planarity Testing

---

[1]The research of this author was supported in part by the National Science Council of the Republic of China.

In the two previous approaches [4,8] of planarity test, the partial subgraph constructed at each iteration is always connected. The st-numbering of Lempel et al's approach further requires that those vertices not added induce a connected subgraph. We adopted a "vertex addition" approach which only requires that those vertices "not added" induce a connected subgraph. Thus, a simple postordering of a depth-first search tree of G suffices.

Let $\mathbf{G_i}$ be the subgraph at the i-th iteration consisting of the first i vertices and those edges among them. In our approach G_i may be disconnected, but the embedding for each biconnected component of G_i, once determined, is never changed.

Assume the given graph G is biconnected and the degree of each vertex is at least 3. To simplify the discussion we use a ***generalized forest representation*** $\boldsymbol{F_i}$ for each G_i. Each node in the forest F_i represents either

(a) an original vertex of G (denoted by a ***v-node***) adjacent to vertices not in G_i.

(b) a biconnected component of G_i whose planar embedding has already been determined (denoted by a ***c-node***).

Let T^* be a rooted depth-first search tree of G. All edges in T^* are called ***tree edges*** and the other edges of G are called ***back edges***. Each tree edge is directed away from the root and each back edge is directed form a node to one of its ancestors. Assume the vertices of G are labeled by a postordering of T^*. Thus, the label of a parent is always larger than those of its children. Sort the neighbors of every vertex in O(*m*) time.

The vertices are added one by one according to their ascending label. Consider the beginning of the i-th iteration of the algorithm at which point vertex i is to be added to G_{i-1}. We need to determine an ordering of edges emanating from vertices in G_{i-1} to i. If a vertex in the current forest has a back edge pointing to i, then there must exist a tree edge directed from i to its root. List the trees of F_{i-1} whose roots are adjacent to i in the current generalized forest as $T_1, T_2, ..., T_r$. Let S be a subset of vertices. Denote by ***G[S]*** the subgraph of G induced on S. Since i is an articulation vertex of the induced subgraph $H_i = G[\{i\} \cup T_1 \cup ... \cup T_r]$, H_i is planar if and only if each $G[\{i\} \cup T_j]$ is planar, j = 1, ..., r. Hence, it suffices to consider the embedding problem of each $G[\{i\} * T_j]$. We shall find an embedding of $G[\{i\} \cup T_j]$ that corresponds to a partial embedding of G. Denote the root of T_j by $\boldsymbol{r_j}$.

Define the ***external degree*** of a vertex at iteration i to be the the number of its neighbors among {i+1,...,n}. Since some of those vertices in G_{i-1} with 0 external degree do not have to be examined for future embedding, we shall apply a **vertex contraction** procedure to eliminate them. The contraction procedure replaces a connected subgraph by a contracted edge, which allows us to recognize and embed the planar graph more efficiently. Let $\boldsymbol{T_j'}$ denote the tree after the vertex contraction from T_j.

At the beginning of the i-th iteration reduce the external degrees of all neighbors of vertex i in F_i by 1. The contraction procedure starts by marking all vertices of T_j that are adjacent to i. Note that a vertex which is not marked at this stage can

become marked later through a contracted edge. There are two types of contraction: contracting vertices on a path and contracting vertices on a cycle. We shall consider the former contraction first. Scan all marked vertices in ascending order. If a marked vertex u is a leaf with external degree 0, then contract u by deleting u and marking parent(u). The edge from parent(u) to i now represents the path: **parent(u)-u-i**.

During the contraction process, if an internal vertex u of T_j becomes a leaf, then all of its descendants must have external degree 0. If several (but not all) children of a vertex u have been contracted when u is scanned, then the edge from u to i represents the connection between i and every vertex in those contracted children subtrees. Each back edge from a vertex u in T_j' to i actually represents the edge connection between i and all contracted vertices in a partial subtree of T_j rooted at u.

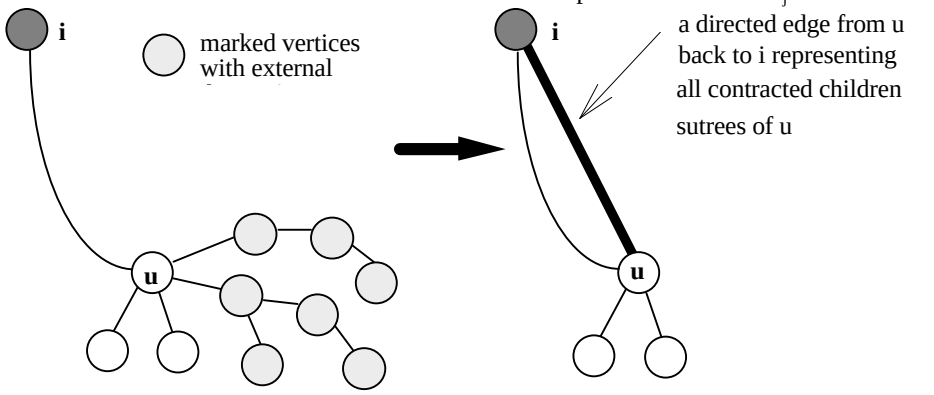

Fig. 2.1 Vertex Contraction

Let u be a marked v-node in T_j' none of its descendents is a contracted c-node. Vertex u is said to be a ***terminal node*** if none of its descendants is marked. We can show that, if G is planar, then there exist at most two terminal nodes in T_j'.

Now consider the embedding at the i-th iteration. Consider two cases: (1) there is only one terminal node u. Let P be the path from r_j to u in T_j'; (2) there are two terminal nodes u and u'. Let P be the unique path from u to u' plus the edge (u',i). We shall describe how to embed path P and those contracted edges in two cases.

2.1 A Special Case of the Embedding

Consider the special case that path P contains only v-nodes. The two cases of path P in the previous section are analogous. Hence, we shall only describe the first case. The back edge (u,i) together with the tree edge (i,r_j) and edges in P form a cycle **C**. Let u_0 (= u), u_1, ..., u_t be the marked nodes in P. We first describe how to construct an internal planar embedding of the biconnected component **H** composed of C and those contracted vertices represented by (u_1,i), ..., (u_t,i). Embed the contracted edges (u_1,i), ..., (u_t,i) on the same side of C. Convert each edge (u_s,i) back to its uncontracted partial subtree in T_j and connect all vertices in the subtree to i. There are many different ways to achieve this. One way is to lay the subtree down on the

plane (namely, specify a left-right children relationships) and to arrange a connecting order based on a preorder traversal.

To represent the biconnected component H in the forest after the embedding of path P, let k_1, ..., k_s be vertices (ordered along the cycle) of C which are adjacent to vertices outside H (namely, either incident to a tree edge or adjacent to a vertex in {i+1,...,n}). Contract all vertices in C-{k_1, ..., k_s} one by one by deleting such a vertex and connect its two neighbors in C. Store k_1, ..., k_s in a circular linked list as the ***representative cycle C(H)*** for H. C' will be denoted by a ***c-node*** in the forest F_i. For future iterations, one can concentrate only on the representative cycle for H.

To maintain the forest structure when more than one T_j is considered, make a copy $\mathbf{i_j}$ of i in each T_j and change all back edges (u,i) from vertices in T_j to (u,i_j). Connect i_j with i using a **virtual tree edge.** Hence, i_j (instead of i) is placed into the cycle C. A tree edge of T_j, once becomes an edge in H, is no longer considered as a tree edge.

2.2 The General Embedding

Now, consider the general case in which path P contains some c-nodes. A node in P which is not the end node is called an ***intermediate node*** of P. In a path P the representative cycle of each intermediate c-node H contains exactly two vertices (called ***high P-vertex h_1*** and ***low P-vertex h_2***, respectively, according to their labels; in case H is an end c-node, then only the high vertex h_1 exists) adjacent to the two tree edges of P. Each v-node in P and each high, low P-vertex of a c-node in P is referred to as a P-vertex. For every non-P-vertex u in the representative cycle define the ***upward*** (resp. ***downward***) ***direction*** as the direction which leads u first to the high (resp. low) P-vertex along the cycle.

In addition to the above vertex contraction procedure, we need to contract vertices in the c-node as follows. Let u be a marked non-P-vertex in a c-node which is not incident to any tree edge nor adjacent to vertices in {i+1,...,n} (a vertex satisfying the latter condition is said to be ***saturated***). If u has only one neighbor in the c-node (which must be the high P-vertex), then delete u and mark the high P-vertex; otherwise, let u_1, u_2 be the two neighbors of u in the representative cycle of H. If one of u_1 or u_2 is a marked non-P-vertex, then contract u by deleting u and connecting u_1, u_2 by an edge. This edge now represents the path: u_1-u-u_2. It is easy to verify that the contracted graph is planar iff the uncontracted one is. Note that an entire c-node could be contracted to a single vertex through this operation. Also, after the contraction, no two saturated vertices can be adjacent to each other.

A c-node is marked if one of the vertices in its representative cycle is marked. A marked c-node is a ***terminal c-node*** if none of its descendants is marked. A marked v-node none of whose descendants are marked is called a ***terminal node.*** A marked non-P-vertex u in a c-node is a ***terminal node*** if either it has non-zero external degree or it is incident to a tree edge. A marked P-vertex in a c-node is a ***terminal node*** if there exists a non-P-vertex which is a terminal node by the above definition.

Obviously, only "one-half" of the c-node can be embedded inside the cycle formed by path P and the edge from the last marked vertex of P to i. It is quite easy to determine which half should be embedded inside by the following Theorem 2.3.

Theorem 2.2. *If G is planar, then there exist at most two terminal nodes in T_j'.*

Consider the contracted T_j'. Let P be a path connecting r_j to a terminal v-node.

Theorem 2.3. *Each marked vertex in a c-node H of P must be adjacent to the high P-vertex h_1. If H is an intermediate c-node in P, then h_1 and h_2 must either be adjacent to each other or to a saturated vertex in C.*

We now describe the embedding of the current graph at iteration i. Refer to the "outside face" as the one containing all vertices in {i+1,...,n}. The reason that these vertices must be contained in one face is that they form a connected induced subgraph of G. Consider two cases:

Case 1. There is only one terminal v-node u.

Let P be the path from r_j to u. Unlike the special case that P contains no c-node (where a unique cycle is formed), we now form two cycles based on P. The **inner cycle** C_1 is used to construct an embedding of the resulting biconnected subgraph; the **outer cycle** C_2 is used to construct the representative cycle for the corresponding c-node. For each intermediate c-node H of P, if $h_1(H)$ is adjacent to $h_2(H)$, define $P_1(H)$ to be the edge $(h_1(H),h_2(H))$, define $P_2(H)$ to be the other half of C(H) from $h_1(H)$ to $h_2(H)$; otherwise, let u(H) be a saturated vertex adjacent to both $h_1(H)$ and $h_2(H)$ (by Theorem 2.3 such a vertex must exist), define $P_1(H)$ to be the set of edges $\{(h_1(H),u(H)),(u(H),h_2(H)\}$, define $P_2(H)$ to be the other half of C(H) from $h_1(H)$ to $h_2(H)$.

Case 2. There are two terminal vertices u and u'.

In case u and u' are on the same c-node, say m, then the outer cycle C_2 consists of the edges (u,i), (u',i) and the path on C(M) from u to u' not passing through $h_1(M)$; the inner cycle C_1 consists of the edges (u,i), (u',i) and the path on C(M) from u to u' passing through $h_1(M)$. Hence, assume u and u' are not on the same c-node. Let m be the least common ancestor of u and u' in T_j'. If m is a v-node then consider the unique path from u to u' and, similar to Case 1, we can form the outer cycle and inner cycle. If m is a c-node, then let v and v' be the vertices connected to u and u' respectively. Then we can form the cycles similar to the above description except that on the c-node M, the outer cycle uses the path on C(M) from v to v' not passing through $h_1(M)$; the inner cycle uses the path from v to v' passing through $h_1(M)$. A necessary condition for the graph to be embeddable is that all vertices in C_1-C_2 are saturated.

We now discuss the time complexity of the PT algorithm. The construction of a depth-first search tree and a postorder traversal takes $O(m+n)$ time. At each iteration, Let the degree of i in G_{i-1} be denoted by $\mathbf{deg_{Gi}(i)}$. Vertex contraction takes time proportional to $deg_{Gi}(i)$. The terminal nodes can be identified by checking the smallest marked vertex in T_j'. In the embedding of a path P starting from a terminal

vertex, each tree edge of P will become a non-tree edge. Summing over all iterations, the number of times a tree edge becomes a non-tree edge is at most O(n).

The key saving in our algorithm is that, in each c-node h, the set of marked vertices with zero external degree in C(H) must be arranged consecutively and hence, can be easily determined. Furthermore, vertices in P_2(H) never need to be traversed in composing the outer cycle. It suffices to find its two end vertices in the linked list. Therefore, besides traversing the tree edges from the terminal vertices (which totaled to be O(n)), the amount of work involved in each iteration is proportional to deg_{G_i}(i). Hence, the complexity of the algorithm is O(m+n).

3. The Maximal Planar Subgraph Algorithm

We now discuss our maximal planar subgraph (MPS) algorithm. It can be viewed as an extension of our planarity Testing (PT) algorithm . The basic idea is to emulate the steps in the PT algorithm. We shall describe one important property of planar graphs that froms the basis of the MPS algorithm. The terminal nodes in the PT algorithm can all be determined uniquely at each iteration. In fact, one can have the following theorem.

Theorem 3.1. *A graph G is planar if, in the above planarity test there are at most two terminal nodes at each iteration.*

By definition, terminal nodes are actually determined after the introduction of certain external edges (these are the edges inhibiting the contraction process). Since the appearance of any external edge in the final subgraph is still in question, we assume all external edges are likely to exist at any iteration. The algorithm will simply "reserve" two terminal node slots to be filled by some candidates later and proceed to the next iteration. This allows the MPS algorithm to proceed more or less like the PT algorithm. Note that as long as there are no more than two terminal nodes created, Theorem 2.2 will guarantee that the final subgraph is planar. The main difference now is that whenever a new edge is included by the MPS algorithm it will exclude a set of edges from further consideration. Such a set can be determined uniquely. Furthermore, no vertex contraction is applied.

The main idea of the MPS algorithm is to traverse the tree towards the root from those vertices adjacent to the new vertex. To accommodate for slot reservation, we need to label the vertices and biconnected components traversed in the algorithm so that each edge will be traversed a constant number of times.

Each tree in the current forest that contains vertices adjcent to the new vertex will produce at most one biconnected component containing that new vertex. The MPS algorithm will assume such a component is already formed (which is actually pending on the determination of terminal nodes) and each vertex of the boundary cycle is labeled with the new vertex. Thus, the next time any vertex, say u, of this component, say C, is traversed two things will happen: (1) vertex u will be identified to be lying on the boundary of the biconnected component formed by the current new vertex; (2) the traversal will continue with the new vertex of C, thus skipping all other vertices of component C.

Let i be the new vertex in the current iteration. Consider a tree with root x in the current forest that contains at least two vertices adjcent to i. Such a tree will create a biconnected component for i at this iteration. We shall associate the edge (i,x) with this component. The MPS algorithm traverses the vertices and edges of the tree as follows. It first picks the neighbor of i with the lowest order, say u, in the tree and traverses the unique path from u to x. Each vertex v on that path will be labeled with (i,x,d), where d is the distance between v and u measured by the number of edges along the path.

Since the postordering guarantees that vertices with lower order will be considered before those with higher orders, those neighbors of i along the path from u to i will be labeled and they will no longer be considered at this iteration. After the unique path from u has been traversed the algorithm will pick the next lowerest unlabeled vertex, say u', adjacent to i and traverse the unique path from u' to i. The traversal will terminate when it encounters a vertex already labeled. The algorithm then picks the next lowerest unmarked vertex and continues until all neighbors of i in this tree have been considered. Everytime a new edge is added to the subgraph some other edges could be deleted. We shall discuss the edge exclusion in the next iteration.

4. Determination of Terminal Nodes

As discussed in the last section, the determination of terminal nodes in any iteration depends on the inclusion of future edges. Thus, everytime a new edge is added to the subgraph G' some candidate edges in previous iterations will be excluded. This process is continued so that for each specific iteration the candidate edges are eliminated until no more than two terminal nodes can appear. Note that at the end of an iteration i all edges in G from i to its descendents have been considered and will be included in the subgraph G' if not already excluded in the process.

Each back edge can form a unique cycle with edges in the tree. Such a cycle is said to be a fundamental cycle.

Let i, j be two integers with $i < j$ satisfying the following conditions:

(a) There is an edge (j,u) between j and a descendent u of i.

(b) Let x be the child of i on the unique tree path from u to i. A component associated with the edge (i,x) was created in the i-th iteration.

(c) i, j are the smallest such integers respectively.

We shall describe the process of edge addition with regard to those edges from j to the descendants of i. The MFS algorithm will traverse the unique tree path from u to i. Each vertex traversed will be labeled. Consider two cases: (i) there is an edge in G' from i to a descendent of u; (ii) no such edge exists in G'. As discussed in the last section each vertex encountered will be labeled with (i,x,d), where d is the distance between v and u measured by the number of edges along the path. Whenever a vertex, say v, encountered has m(v) = (i,x,d') we check m(i,x). In the discussion below we shall describe the exclusion of candidate edges. Consider the following three cases:

(1) m(i,x) = 0: (j,u) is the first external edge added to G' with u being a descendant of i. We shall give vertices u and v special names, u_1 and v_1, respectively and

change m(i,x) to be 1. A biconnected component for i will be formed as depicted in the left diagram of Figure 4.1, where the triple thick lines indicate that there is a biconnected component for i whose outer boundary has not yet been fixed. The MPS algorithm will then traverse the unique tree path from v to i (thus, edges on this path are traversed twice).

(2) m(i,x) = 1: some biconnected component for i has already been discovered.

We change m(i,x) to 2 and continue the traversal until a vertex, say v', labeled with (j,d") is encountered, where $j \neq i$. The MFS algorithm will then visit i, skipping vertices on the unique tree path between v' and i. Consider the following two subcases:

(a) If $v' = v_1$, the unique path from v_1 to i (respectively, v) is "on" the outer boundary of the biconnected component for i (respectively, for j). However, the remaining part is still not fixed as depicted in the middle diagram of Figure 4.1.

(b) If $v' \neq v_1$, then the unique path from v to v_1 is on the outer boundary of the biconnected component for i as well as the component for j and all the remaining part is not fixed as depicted in the right diagram of Figure 4.1.

We shall give vertices u and v special names, u_2 and v_2. Note that if a part is wrapped inside the biconnected component for j, then it is immaterial which path in that part should be on the outer boundary of the component for i. Furthermore, only descendents of v will remain as viable candidates and we can eliminate the external edges for all vertices wrapped inside the biconnected component for j.

Finally, the path from u_1 through v, v_2, i to j has its vertices divided into two segments: those in between u_1 and v_2 and those in between v_2 and j (even though the exact path from v_2 to i has not yet been fixed). Such a partition has the following effect: if any vertex in a segment is first encountered through a tree traversal when an edge (j',u') is added where u' is a descendant of i and j' is an ancester of j, then the other segment will be "wrapped" inside by the edge (j,u_2).

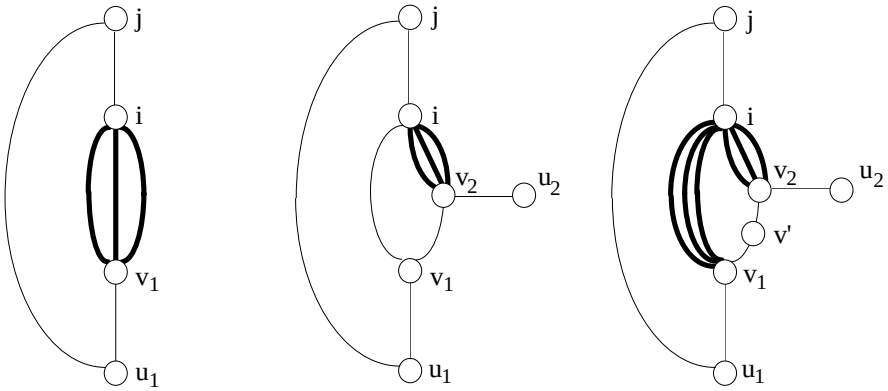

Fig. 4.1. The change of mark for the edge (i,x) from 0 to 2

(3) m(i,x) = 2: let the label of i be (j,d).

This is the situation that, at the j-th iteration some vertex (which is then named u_2) adjacent to j caused m(i,x) to be changed from 1 to 2. By the postordering of the depth-first-search tree, we can argue that v must be an ancestor of either v_1 or v_2.

Hence v must be inserted into the segment between v_1 and v_2. Thus, this segment is futher divided. We call such a process a ***refinement***. Futhermore, if there were similar vertices, say v', being inserted in between v_1 or v_2 before v, then v' cannot be an ancestor of v. Hence, the vertices inserted into the segment between v_1 and v_2 can be stored in two stacks, one for the ancestors of v_1 the other for the ancestors of v_2. Thus, each such insertion takes only constant time. At the end of the insertion process, the MFS algorithm will then pick the next neighbor of j and continue with the traversing process.

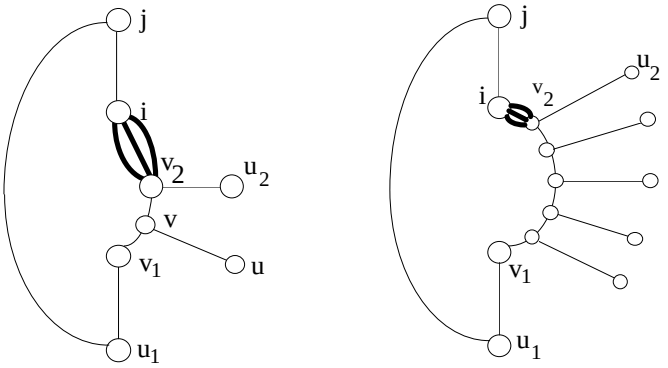

Fig. 4.2. The refinement of the outer boundary of a biconnected component

Each time such a refinement occurs, However, the argument at the end of (2) with regard to the wrapping effect of the edge (j,u) still holds except that there are more segments now to be wrapped. If a traversal does not run into any intermediate vertex of a biconnected component, then this component can be regarded as an "edge", one that connects the lower P-vertex to its high P-vertex. Hence, it is immaterial whether the internal vertices of such a component are fixed or not. If a traversal encounters an intermediate vertex, then the outer boundary of this component will be further refined and the discussion of the earlier part of this section applies. In case the components are "nested", the refinement will be done recursively.

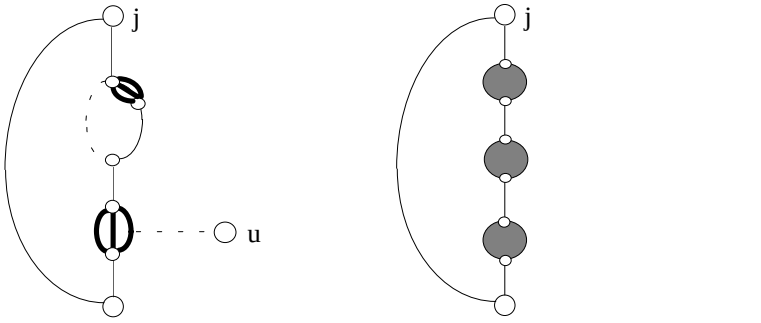

Fig. 4.3. Nested biconnected components

5. Complexity Analysis

The argument for the linear complexity is mainly based on the fact that the number of times an edge is traversed is a constant. First of all, consider each tree edge. The first time a tree edge e is traversed is trickled by the addition of a back edge to the subgraph G'. Whenever edge e is wrapped inside a biconnected component, it is eliminated from further consideration. Otherwise, e will be traversed the second time when a potential biconnected component is formed for some edge (i,x). The third time the edge is traversed (if not already wrapped inside a component) is when it is on the outer boundary cycle for some component.

Now consider each back edge. The first time a back edge is added to the subgraph G' a biconnected component is formed and this edge is on the outer boundary cycle. Afterwards this edge will be traversed only when it is wrapped inside a component.

Finally, when a biconnected component is regarded as an "edge", the number of times such an edge will be traversed is also a constant, based on an argument similar to the beginning paragraph of this section. Hence, the time complexity of the MPS algorithm is linear.

REFERENCES

1. K. S. Booth and G. S. Lueker, *Testing the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms,* **J. Comput. Syst. Sci.** 13(1976), 335-379.
2. J. Cai, X. Han and R. T. Tarjan, *An O*(*mlogn*)*-time algorithm for the maximal planar subgraph problem*, **SIAM J. Comput.** 22(1994), 1142-1162.
3. G. Di Battista and R. Tamassia [1989], *Incremental planarity testing,* in Proc. 30th Annual IEEE Symposium on Foundation of Computer Science, 436-411.
4. J. E. Hopcroft and R. E. Tarjan, *Efficient planarity testing,* **J. Assoc. Comput. Mach.** 21(1994), 549-568.
5. W. K. Shih and W. L. Hsu, *A simple test for planar graphs,* 1993, submitted.
6. R. Jakayumar, K. Thulasiraman and M. N. S. Swamy, *An O*(n^2) *algorithms for graph planarization*, **IEEE Trans. CAD** 8(1989), 257-267.
7. J. A. La Poute, *Alpha-Algorithms for incremental planarity testing*, **STOC** 1994, 706-715.
8. A. Lempel, S. Even and I. Cederbaum, *An algorithm for planarity testing of graphs*, **Theory of Graphs**, ed., P. Rosenstiehl, Gordon and Breach, New York, 1967, 215-232.
9. W. Wu, *On the planar imbedding of linear graphs,* **J. Systems Sci. Math. Sci.** 5, 290-302, 1985.