An Efficient Implementation of the PC-Tree Algorithm of Shih & Hsu's Planarity Test

Wen-Lian Hsu, hsu@iis.sinica.edu.tw

Institute of Information Science, Academia Sinica, Taipei, Taiwan, ROC

Abstract

In Shih & Hsu [9] a simpler planarity test was introduced utilizing a data structure called PC-trees (generalized from PQ-trees). In this paper we give an efficient implementation of that linear time algorithm and illustrate in detail how to obtain a Kuratowski subgraph when the given graph is not planar, and how to obtain the embedding alongside the testing algorithm. We have implemented the algorithm using LEDA and an object code is available at http://qa.iis.sinica.edu.tw/graphtheory/. The main part of the implementation is devoted to the treatment of the *C*-nodes that represent those 2-connected components.

Keywords: algorithm, planar graphs, Kuratowski subgraph, PQ-tree, PC-tree, embedding

1. Introduction

Given an undirected graph, the planarity test 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 test was first established by Hopcroft and Tarjan [5] based on a "*path addition* approach." A "*vertex addition* approach", originally developed by Lempel, Even and Cederbaum [6], was later improved by Booth and Lueker [1] (hereafter, referred to as B&L) to run in linear time using a data structure called a "*PQ*-tree". Both of these approaches are quite complex. Furthermore, both approaches use separate algorithms for recognition and embedding (Chiba et al [4]). Several other approaches have also been developed for simplifying the planarity test (see for example [3,9,11,14]) and the embedding algorithm [2][8]. Shih and Hsu [10] developed a very simple linear time test and later [11] implemented it based on PC-trees. The latter is referred to as S&H hereafter. When the given graph is not planar, their algorithm immediately produces explicit Kuratowski subgraphs. Furthermore, the recognition and embedding are done simultaneously in their algorithm. The algorithm in [10] has been referred to as the simplest linear time planarity test by Thomas in his lecture notes [13].

In Section 2 we review the basic notations used in the S&H algorithm. Section 3 outlines the streamlined algorithm for the simple case (without biconnected components). The notation of PC-trees is introduced in Section 5. Section 6 discusses the algorithm for the general case.

2. The Edge-Addition approach of the S&H Algorithm

Let n be the number of vertices of the graph G. Construct a depth-first search tree T for G. Note that every non-tree edge of G must be a back edge from a vertex to one of its ancestors. To simplify our discussion, assume the given graph G is biconnected. This is certainly not a restriction since we can split the graph into biconnected components along articulation vertices, which can be identified in the depth-first-search tree. Let 1, ..., n be the order resulting from a postorder traversal of T. So the order of a child is always less than that of its parent. Denote the subtree of T with root i by T_i. Initially, we include all edges of T, namely the depth-first-search tree, in the embedding. Then, at iteration i, we add all back edges from the descendants to node i and update the embedding. Whenever a 2-connected subgraph is created, we use a subset of vertices in its boundary cycle as representatives to be used for future embedding. Hence, our algorithm can be considered as an "edge-addition" approach (it was originally called a vertex-addition approach in [9]). The embedding of each 2-connected component is temporarily stored so that, at the n-th iteration, when the graph is declared planar, a final embedding can be constructed by tracing back and pasting the internal embedding of each 2-connected component along its corresponding boundary cycle.

Denote the largest neighbor of a node i by $h(i)$. Sort the children of each node of T according to the ascending order of their labels. At each iteration i, we consider the embedding of the back edges from the descendants to i and revise the tree accordingly. Denote the revised tree at the end of iteration i by T^i.

To facilitate the description of notations used in this paper, we divide our discussion into two parts as in the original S&H paper. In the first part, we consider the iteration in which a biconnected component is formed "for the first time" in Section 3. The general case will be left to Section 5.

3 The Simple Case

Let i be the first iteration in which there is a back edge from a descendant i' to node i. Then (i', i) together with the unique path from i to i' in T form a cycle. Thus, this is the first iteration a 2-connected component is formed. We shall describe which vertices should be embedded inside such a component and which should be located on its boundary. It suffices to describe the embedding of i with each child subtree independently. Hence, consider a child subtree T_r of i (with root r) that has a back edge to i. Note that there will be at most one biconnected component formed from this subtree at the end of this iteration. The following definitions will be useful for the description of the algorithm.

Definition 3.1. Classify the nodes in T_r into four types: A node is type 1 if it has no back edge; it is type 2 if it has a back edge to i and no other back edge; it is type 3 if it has a back edge to i and a back edge to a node > i; and it is type 4 if all of its back edges are to nodes > i.

Note that a leaf in T_r cannot be type 1, otherwise the parent of this leaf would be an articulation vertex.

Definition 3.2. A leaf in T_r is full if it is type 2; it is partial if it is type 3; it is empty if it is type 4. A subtree T_v is full if it contains only type 1 and type 2 nodes; T_v is partial if it either contains a type 3 node, or contains both a type 2 node and a type 4 node; T_v is empty if it contains only type 1 and type 4 nodes. A node v is full (respectively, partial, empty) if T_v is full (respectively, partial, empty). Define a terminal node in T_r to be a partial node whose children are either full or empty.

Assume the graph is planar. We have the following important properties from [11]:

(a) *The parent of a partial node is partial.*

(b) *A partial node must contain a terminal node as a descendant.*

(c) *There are at most two terminal nodes in T_r.*

(d) *Any node v with two descendant terminal nodes satisfies $h(v) \leq i$.*

We use a tree traversal algorithm in Figure 1 to identify all full nodes and partial nodes. During this process, a node could first receive a label as partial and then a label as full. Note that empty nodes are not traversed at all so they do not receive a mark or a label.

```
Labeling-Algorithm
1.    Label all type 2 leaves full; label all type 3 leaves partial
2.    Process type 2 nodes in the ascending order starting from the lowest-indexed full leaf
3.    When a node is labeled as a full node, label its parent (if still unlabeled) partial
4.    If all children of a node $v$ become full and $h(v) \leq i$, label $v$ full.
5.    If the next type 2 node to be processed has an index larger than a node $v$ that is labeled partial, label the parent of $v$ partial.
6.    Repeat steps 3, 4, 5 until all type 2 nodes have been processed. Label all ancestors of partial nodes in $T_r$ partial.
```

Figure 1. The Labeling-Algorithm

When the above labeling algorithm terminates, we get all full nodes and partial nodes labeled correctly. Because of the postorder of the indices, the first partial node *u* encountered by the algorithm must be the lowest indexed partial node, and hence a terminal node. Let P_u be the unique path from *u* to root *r*. Let Q_i be the set of partial nodes. If node *u* is the only terminal node, then $Q_i = P_u$ and we obtain all partial nodes by tracing path P_u. Otherwise, let u' be the lowest indexed node in $Q_i - P_u$, then u' is another terminal node. Nodes on the unique path $P_{u'}$ from u' to root *r* are partial, and $Q_i = P_u \cup P_{u'}$. Hence, we obtain all partial nodes by tracing both P_u and $P_{u'}$. An illustration of these properties in the case of two terminal nodes is shown in Figure 2.

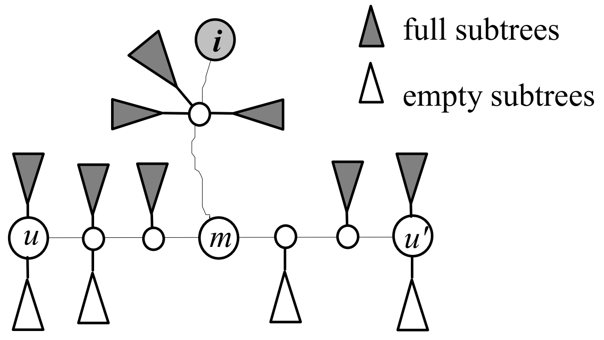


Figure 2. The full subtrees and empty subtrees

Another important feature of the S&H algorithm is its characterization of the boundary of the resultant biconnected component. In Case 1, let u' be the last node along P_u from u to r that has an empty child. In Case 2, let u' be the second terminal node. In either case, let P be the unique path in T_r from u to u'. Then Path P must be on the boundary of any 2-connected components formed by node i and nodes in the subtree T_r. In Section 5, we demonstrate how to maintain a tree-like structure by replacing each biconnected component with a C-node.

4. Identifying Kuratowski Subgraphs Homeomorphic to $K_{3,3}$

We illustrate how to obtain Kuratowski subgraphs in case the given graph is not planar. There are many situations in which our program can detect non-planarity. A detailed case analysis can be found in our program. As an example, let us consider the following situation.

In Section 3, condition (c) dictates that there can be at most two terminal nodes. Suppose this condition is violated and there are three terminal nodes, say, i_1, i_2 and i_3 (none of them can be a descendant of the other). Then each has a descendant (could be itself) with a neighbor larger than i. For example, consider i_1. If i_1 has a neighbor $> i$, then let this neighbor be t_1. Otherwise, i_1 has an empty child. Every leaf of this child must have label $> i$. Choose a leaf and a back edge from this leaf to a neighbor $t_1 > i$. We can similarly choose t_2 and t_3 for i_2 and i_3, respectively. Note that t_1, t_2 and t_3 must lie on the unique path from i to the root n. Let the medium of these three (not necessarily distinct) neighbors be t.

We can choose three node-disjoint paths P_{tm} from t to t_m and through the unique tree path from t_m to i_m for m = 1, 2 and 3 respectively.

Let the smallest of the three least common ancestors of $\{i_1, i_2\}$, $\{i_1, i_3\}$ and $\{i_2, i_3\}$ be w. Let P_{wm} be the three unique tree paths from w to i_m for m = 1, 2 and 3, respectively.

For each i_m, m = 1, 2, 3, let s_m be a leaf (could be itself) in one of its full child; let P_{im} be the path from i to s_m and through the unique tree path to i_m.

It is not difficult to verify that these nine paths, P_{tm}, P_{wm} and P_{im}, m = 1, 2, 3 are internal node-disjoint. Therefore, a subgraph homeomorphic to $K_{3,3}$ can be found as shown in Figure 3, where the dotted lines denote these paths.

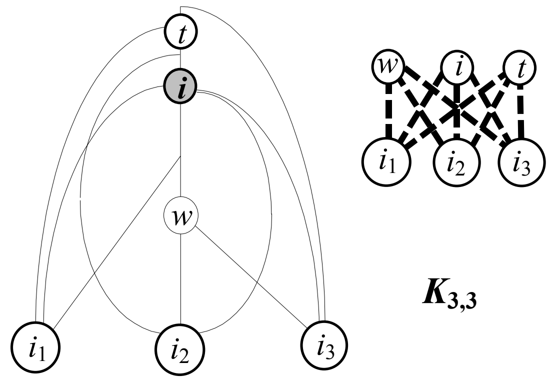


Figure 3. A forbidden subgraph $K_{3,3}$

5. PC-Trees

To facilitate the implementation of the general case, in particular, the representation of biconnected components, the notion of PC-trees was introduced in [11]: a tree is a PC-tree if its nodes can be divided into two types: *P*-nodes and *C*-nodes, where the neighbors of a *P*-node (denoted by a circle) can be permuted arbitrarily and the neighbors of a *C*-node (denoted by a double circle) must observe a cyclic order, which can only be reversed. We use a PC-tree to represent the partial embedding of the planar graph in which a *P*-node denotes a regular vertex of the graph and a *C*-node denotes its corresponding biconnected component with its representative vertices (to be defined in Section 5.1) as children.

5.1 Creation of *C*-Nodes

In Section 3 we described how to identify full nodes and partial nodes as well as the boundary path of the biconnected component. We will embed all full nodes inside the component and leave the empty nodes outside. For each node j on the unique path P, define its *new set of children* as children that are empty nodes. Note that empty nodes must be embedded outside the biconnected component. Define the *essential nodes* on the boundary cycle to be those that have a back edge to a node greater than i or has at least one empty child. Since these nodes must be in P, they are independent of the selection of the exact boundary cycle (there could be many choices) of the component. The essential nodes are the only ones relevant to future embedding.

Define the *representative boundary cycle* (*RBC*) of such a 2-connected component to be a cycle iPi, formed by node i, and those essential nodes whose cyclic order follows their original order in path P. Define node *i* to be the *head* of this RBC. The RBC will be stored as a circular doubly linked list. To distinguish it from the original edges of the graph, we refer to the connections on the RBC as *links*.

The internal embedding will be discussed in Section 6.4. To maintain a tree-like structure for the current embedding, we represent the 2-connected component by a *C*-node, say, *w*, whose parent is *i* and whose children are the essential nodes as shown in Figure 4. Define the two *end nodes* of *w* to be the two neighbors of *i* in its RBC; define *head*(*w*) = *i*. Since *w* has no back edge, *h*(*w*) = *i*.

A *C*-node can only be adjacent to *P*-nodes. It can be easily verified that this property holds throughout the algorithm.

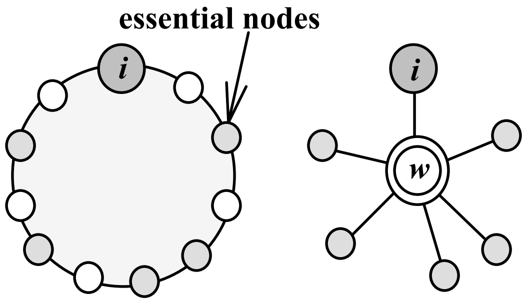


Figure 4. The representation of a *C*-node

5.2 Difference between the PQ-Tree and the PC-Tree Approaches

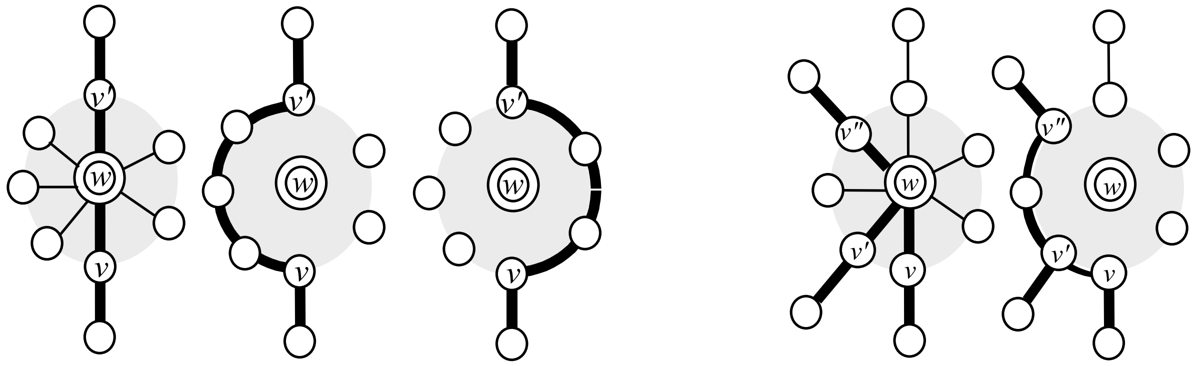
The way we adopt PC-trees in our planarity test is entirely different from B&L's application of PQ-trees in Lempel, Even and Cederbaum's planarity test [6]. B&L used PQ-trees to test the consecutive ones property of all nodes adjacent to the incoming node in their vertex addition algorithm. The leaves of their PQ-trees are exactly those nodes adjacent to the incoming node. Internal nodes of the PQ-trees are not the original nodes of the graph. They are there only to keep track of feasible permutations. Whereas in our approach, every *P*-node is an original node of the graph, every *C*-node represents a biconnected component in the partial embedding, and nodes adjacent to the incoming node can be scattered anywhere, both as internal nodes and as leaves in our PC-tree. Thus, in our approach, a PC-tree faithfully represents a partial planar embedding of the given graph and is a more natural representation. Another difference is that in order to apply PQ-trees in B&L's approach, there has to be a preprocessing step of computing the s-t numbering besides the depth-first search tree. This step could create a problem when one tries to apply PQ-trees to find maximal planar subgraphs of a general graph.

Although PC-trees provide a tree-like representation and is easy to handle conceptually, we must avoid the frequent change of parent pointers in a linear time implementation. Therefore, similar to B&L's treatment of *Q*-nodes [1], we adopt the strategy of borrowing parent pointers through the siblings in its boundary cycle and keep parent pointers only for the two endmost nodes, namely, siblings of the head node. But, this is equivalent to traversing along the RBC to identify the parent.

This concludes our discussion on the algorithm in the first iteration in which some back edges emerge. At the end of this iteration, we have *C*-nodes in the revised tree T^i.

6. The General Case

In this section we discuss the case where there are *C*-nodes in the current tree. Again, assume the graph is planar. We denote the current iteration by *j* and the child subtree by T_r. In many ways a *C*-node can be treated just like a *P*-node. Properties (a) through (d) and most of the arguments in Section 3 will stand without much change as long as the tree paths are interpreted correctly: a tree path through a *C*-node *w* in T_r should be interpreted as a path using a boundary path from *v* to *v′* in its original 2-connected component, where *v* and *v′* are the neighbors of *w* in the path; two paths merge into one at a *C*-node *w* should be interpreted as two paths merge along the boundary into another path emanating from the boundary path. These are illustrated in Figure 5.



(i) Interpreting a path through *w* from *v* to *v′* (ii) Two paths into *v*, *v″* and out from *v′*

Figure 5. The interpretation of tree paths through a *C*-node *w*

Denote the current iteration by *j* and the child subtree by T_r. The definition of a terminal node remains the same; however, the computation of full nodes now has to involve *C*-nodes, which is discussed in Section 6.1. The discussion regarding the two cases in Section 3 needs to be modified. We will follow the notations in Section 3. Let *j* be the current iteration. Let *u* be the first terminal node identified. Let Q_j be the set of partial nodes. If node *u* is the only terminal node, then $Q_j = P_u$ and we obtain all partial nodes by tracing path P_u. Besides satisfying properties (a), (b), (c), (d), we have the following property that all C-nodes can be flipped correctly to one side:

(e) *If u is a C-node, then the set of full children plus head(u) are consecutive in its RBC. Consider two cases:*

1. *If only one neighbor of head(u) in the RBC is full, then for each internal C-node v of P_u, let v_1 and head(v) be its two neighbors in P_u, then the full children of v are consecutive in the RBC and each of v_1 and head(v) has exactly one full neighbor in the RBC.*
2. *If both neighbors of head(u) in the RBC are full, then every ancestor v of u in P_u satisfies that $h(v) \leq i$ and its children not on P_u are full.*

On the other hand, if $Q_j - P_u \neq \varnothing$, let u' be the lowest indexed node in $Q_j - P_u$. Then u' is another terminal node. Nodes on the unique path $P_{u'}$ from u' to root r are partial and $Q_j = P_u \cup P_{u'}$ and we can obtain all partial nodes by tracing both P_u and $P_{u'}$. In addition, property (e) must hold to ensure all partial C-nodes can be flipped correctly to one side.

6.1 Computation of Full *C*-Nodes

Recall that a node v is full at the j-th iteration if T_v contains only type 1 and type 2 nodes. For a C-node w to be a full node, all of its children must be full. This is checked efficiently as follows. Let the RBC for w be C_1 with $head(w) = i$. Let the two end nodes of C_1 that have parent pointers be p and q. When a node v of C_1 becomes full it will pass this message to its two neighbors in C_1. Also, v will check if there is any message passed from each of its two neighbors. The idea is to form blocks of contiguous full nodes in C_1 by merging smaller blocks so that eventually, if both p and q belong to the same block, we can declare that w is a full node. The details are described in Figure 6.

For a block of contiguous full children with end nodes b_1 and b_2 in C_1, we define the end-node function $e(\cdot)$ on b_1 and b_2 as follows: $e(b_1) = b_2$, and $e(b_2) = b_1$. In case b_1 equals b_2, the block contains only one node. We use the Block-Computation-Algorithm in Figure 6 to compute the blocks of full nodes in C_1. In case we have $e(p) = q$ or $e(q) = p$, then all children of w are full and w is also full.

```
Block-Computation-Algorithm
A new full node $v$ in C1 is computed. Let $v_1$, $v_2$ be the two neighbors of $v$ in $C_1$. Consider the following cases:
Case 1. $v_1$ is full, $v_2$ is not full
        Let $B_1$ be the block with $v_1$ as an end node and $e(v_1)$ as the other end node
        Compute a new block $B_1' \leftarrow B_1 \cup \{v\}$; $B_1'$ now has $v$ and $e(v_1)$ as its two end nodes;
        Define $e(v) \leftarrow e(v_1)$ and $e(e(v_1)) \leftarrow v$
Case 2. $v_2$ is full, $v_1$ is not (similar to the above with 1 replaced by 2)
Case 3. Both $v_1$ and $v_2$ are full
        Let $B_2$ be the block with $v_2$ as an end node and as the other end node
        Compute the new block $B_v \leftarrow B_1 \cup \{v\} \cup B_2$; $B_v$ now has $e(v_1)$ and $e(v_2)$ as its two end nodes;
        Define $e(e(v_1)) \leftarrow e(v_2)$ and $e(e(v_2)) \leftarrow e(v_1)$
Case 4. Neither $v_1$ nor $v_2$ is full
```

```
      Form a new block $B_v = \{v\}$ containing one node
```

Figure 6. The Block-Computation-Algorithm

If the graph is planar and a *C*-node *w* is partial, then we have either (i) only one block *B* of contiguous full children with one end node (*p* or *q*) of C_1 being the end node of the block *B*; or (ii) There are two blocks in which one block B_1 has *p* as an end node, and the other block B_2 has *q* as its end node.

By making use of the full *C*-node computation above, the General-Labeling-Algorithm in Figure 7, revised from the labeling-algorithm in Section 3, allows us to partition nodes of T_r into full, partial and empty nodes. Hence, we can compute the unique path *P* from a terminal node *u* to *u′*.

```
General-Labeling-Algorithm
1.  Label all type 2 leaves full; label all type 3 leaves partial.
2.  Process type 2 nodes in the ascending order starting from the lowest-index full leaf.
3.  When a node $v$ is labeled as a full node, consider two cases:
      Case 1. The parent of $v$ is a $P$-node. Then label its parent (if still unlabeled) partial.
      Case 2. The parent of $v$ is a $C$-node $w$ with RBC $C_1$. Then compute a block of full children containing $v$ using the Block-Computation-
          Algorithm in Figure 6. If this block contains all children of $w$, then label $w$ full. Otherwise, if $v$ is an end node of $C_1$, then label $w$
          (if still unlabeled) partial.
4.  If all children of a node $v$ become full and $h(v) \le i$, label $v$ full.
5.  If the next type 2 node to be processed has an index larger than a node $v$ that is labeled partial, label the parent of $v$ partial.
6.  Repeat steps 3, 4, 5 until all type 2 nodes have been processed. Label all ancestors of partial nodes in $T_r$ partial.
```

Figure 7. The General-Labeling-Algorithm

In terms of complexity, this section contains the idea for the major saving of our implementation because it allows us to treat a *C*-node like a *P*-node. Even though not every child of a *C*-node has an explicit parent pointer, by virtue of the property that full nodes are contiguous in its RBC, the amount of time spent to determine whether this *C*-node is full or partial is still proportional to the number of full children it has. Thus, the general case is not more difficult to handle than the simple case.

6.2 A Kuratowski Subgraph Homeomorphic to K_5

In Section 4 we illustrate how to identify a subgraph homeomorphic to $K_{3,3}$ in one situation. If there is a violation to condition (c), it is possible to identify a subgraph homeomorphic to K_5 in the following situation (this happens when the current tree contains *C*-nodes). We follow the notations in Section 4. Let the three terminal nodes i_1, i_2, and i_3 be neighbors of the same *C*-node *w* of *i*. Suppose the three neighbors t_1, t_2, and t_3 satisfy that $t_1 = t_2 = t < t_3$. Then we would obtain the following ten internal node-disjoint paths.

For each i_m, $m = 1, 2, 3$, let s_m be a leaf (could be itself) in one of its full children; let P_{im} be the path from i to s_m and through the unique tree path to i_m.

Let P_{wm} be the three paths from i_m to i_{m+1} around the RBC for $m = 1, 2$, and 3, respectively, where we let $i_4 = i_1$.

Let P_{tm} be the three node-disjoint paths from t to t_m and through the unique tree path from t_m to i_m for $m = 1, 2$, and 3, respectively.

Finally, let P_{it} be the unique tree path from i to t. It is not difficult to verify that the above ten paths are internally node-disjoint paths and we would get a subgraph homeomorphic to K_5 as illustrated in Figure 8.

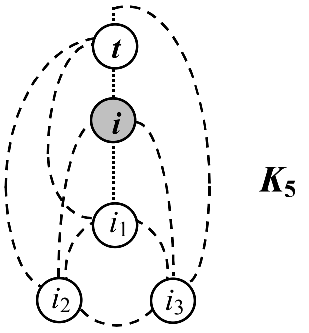


Figure 8. A forbidden subgraph homeomorphic to K_5

6.3 Computation of the RBC

By unwinding the RBC of each C-node in the path P, we are able to determine the "actual" RBC of the partial C-node w of node j. Consider the following two cases:

Case 1. There is only one block B of contiguous full children. One end node, say, p, of C_1 is the end node of the block B. Let t be the other neighbor of $e(p)$ in $C_1 - B$. Then path P will pass through the two partial nodes t and i. The remaining empty nodes of C_1 will form another contiguous block B'. We call the path from t through nodes in block B to i the *inner path* of w, and the path from t through nodes in block B' to i the *outer path* of w.

Case 2. There are two blocks in which one block B_1 has p as an end node, and the other block B_2 has q as an end node. Let t_1 be the other neighbor of $e(p)$ in $C_1 - B_1$ and t_2 be the other neighbor of $e(q)$ in $C_1 - B_2$. Then both t_1 and t_2 are partial, and w must be the unique terminal node of T_r. The remaining empty nodes of C_1 will form another contiguous block B'. We call the path from t_1 through nodes in B_1, node i, nodes in B_2 to t_2 the *inner path* of w, and the one from t_1 through nodes in B' to t_2 the *outer path* of w.

Define the *inner path* (respectively, *outer path*) of the unique path P to be the path containing all P-nodes in P together with all inner paths (respectively outer paths) of C-nodes in P. Thus, the outer path of P consists of those P-nodes in P plus the outer paths of C-nodes in P. Note that the essential nodes must reside on the outer path of P.

6.4 The Embedding Problem

We continue with the embedding of the component for the C-node w we began in Section 6.2. Note that this new component will merge the individual components of C-nodes in path P. The new C-node will take over the remaining essential nodes of C-nodes in P. By fixing the embedding of C-nodes in path P, we can use the inner path P' of P to construct the internal embedding for the remaining part of w. By combining this latter embedding with the embedding of C-nodes in path P, we would obtain the embedding for w. Now, form a tree embedding for P' together with all full subtrees of w by flipping all full subtrees to the "left" of path P'. Embed the edges from j to all its neighbors in this embedding by connecting j to those P-nodes directly and to the essential nodes of the C-nodes in the full subtrees.

Using this method, we see that a biconnected component N_1 formed in one iteration could be merged into another biconnected component N_2. In this case, part of the boundary (the outer path) of N_1 will become part of the boundary of N_2. By retaining the imbedding of N_1, we embed the remaining (internal) part, say, N_3, of N_2 using the inner path of N_1. Thus, the imbedding of N_2 can be formed by gluing the embedding of N_1 to N_3 along that inner path.

Rather than storing the embedding of N_2, we store the embeddings of N_1 and N_3. The embedding of each biconnected component formed by the algorithm is stored incrementally. This would take linear space in total since each edge appears in at most two components. The containment relationships of the 2-connected components created during the algorithm can be recorded by a tree. At the end of the algorithm, when G is verified to be planar, we can form an embedding of G by backtracking through the iterations and gluing these embeddings along the respective inner paths.

6.5. Summary of the Recognition Algorithm

In this section, we give a pseudo-code in Figure 9 summarizing the recognition algorithm.

```
Planarity-Test
1. Find a depth-first-search tree T
2. Obtain a postorder from T for the vertices
3. Order the neighbors of each vertex into ascending order
```

```
4.  For $i$ = 1 to $n$ do
      For each child $r$ of $i$ do
        Form a 2-connected component, if any, for the child subtree $T_r$
          Compute the full nodes using the General-Labeling-Algorithm
          Let the smallest partial node, $u$, be the first terminal node
          Identify the unique terminal path $P$
            Case 1. $u$ is the only terminal node. Let the unique path $P$ be the path from $u$ to $r$
              If condition (6.1.1) is not satisfied, then identify a Kuratowski subgraph
            Case 2. There is one other terminal node $u'$.
              Let $P_{u'}$ be the path from $u'$ to $r$ in $T_r$ and label all nodes in $P_{u'}$ partial
              Let $m$ be the least common ancestor of $u$ and $u'$
              Let $P$ be the unique tree path from $u$ to $u'$; let $P'$ be the unique tree path from $m$ to $r$
              If either condition (6.1.2) or (6.1.3) is not satisfied, identify a Kuratowski subgraph
            Case 3. There are more than 2 terminal nodes. Then identify a Kuratowski subgraph.
          Compute the inner path and the outer path of $P$ as in Section 6.3.
          Locate the essential nodes of this biconnected component on the outer path of $P$
          Form a new $C$-node $w$ with parent $i$ and the essential nodes as children; keep parent pointer only for its two endmost children;
          form a circular linked list with node i and the above essential nodes according to their order in path $P$; $head(w) \leftarrow i$; $h(w) \leftarrow i$
        End;
      End;
    End;
5.  Declare the graph planar
```

Figure 9. The Planarity-Test

7. Acknowledgement

We would also like to thank the National Science Council for their generous support under Grant NSC 91-2213-E-001-011.

References

1. K. S. Booth and G. S. Lueker [1976], *Testing the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms*, **J. Comput. Syst. Sci.** 13, pp. 335-379.
2. J. Boyer and W. Myvold [1999], *Stop minding your P's and Q's: A simplified O(n) embedding algorithm*, Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 140-149.
3. J. Cai, X. Han and R. E. Tarjan [1993], *An O(mlogn)-time algorithm for the maximal planar subgraph problem*, **SIAM J. Comput.** 22, pp. 1142-1162.

4. N. Chiba, T. Nishizeki and S. Abe and T. Ozawa [1985], *A linear algorithm for embedding planar graphs using PQ-trees*, **J. Comput. Syst. Sci.** 30, pp. 54-76.
5. J. E. Hopcroft and R. E. Tarjan [1974], *Efficient planarity testing*, **J. Assoc. Comput. Mach.** 21, pp. 549-568.
6. Lempel, S. Even and I. Cederbaum [1967], *An algorithm for planarity testing of graphs*, **Theory of Graphs**, ed., P. Rosenstiehl, Gordon and Breach, New York, 215-232.
7. K. Mehlhorn [1984], *Graph algorithms and NP-completeness*, **Data Structure and Algorithms** 2, pp. 93-122.
8. K. Mehlhorn and P. Mutzel [1994], *On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm*, **Algorithmica** 16, No. 2 (1996) 233--242.
9. J. Small [1993], *A unified approach of testing, embedding and drawing planar graphs*, Proc. ALCOM International Workshop on Graph Drawing, Sevre, France.
10. W. K. Shih and W. L. Hsu, "A simple test for planar graphs," Proceedings of the International Workshop on Discrete Math. and Algorithms, University of Hong Kong, (1993), 110-122.
11. W. K. Shih and W. L. Hsu, "A new planarity test," **Theoretical Computer Science 223**, (1999), pp. 179-191.
12. H. Stamm-Wilbrandt [1993], *A simple linear-time algorithm for embedding maximal planar graphs*, Proc, ALCOM International Workshop on Graph Drawing, Sere, France.
13. R. Thomas [1997], *Planarity in linear time – Lecture Notes*, Georgia Institute of Technology, http://www.math.gatech.edu/~thomas/.
14. S. G. Williamson [1984], *Depth-first search and Kuratowski subgraphs*, **J. ACM** 31, pp. 681-693.