# 1

# PQ Trees, PC Trees, and Planar Graphs

Wen-Lian Hsu
*Academia Sinica*

Ross M. McConnell
*Colorado State University*

## 1.1 Introduction

A graph is *planar* if it is possible to draw it on a plane so that no edges intersect, except at endpoints. Such a drawing is called a *planar embedding.*

Not all graphs are planar: Figure 1.1 gives examples of two graphs that are not planar. They are known as $K_5$ , the complete graph on five vertices, and $K_{3,3}$, the complete bipartite graph on two sets of size 3. No matter what kind of convoluted curves are chosen to represent the edges, the attempt to embed them always fails when the last of the edges cannot be inserted without crossing over some other edge, as illustrated in the figure.

There is considerable practical interest in algorithms for finding planar embeddings of planar graphs. An example of an application of this problem is where an engineer wishes to embed a network of components on a chip. The components are represented by wires, and no two wires may cross without creating a short circuit. This problem can be solved by treating the network as a graph and finding a planar embedding of it. Planar graphs play a central role in geographic information systems, and in many problems in computational geometry.

The study of planar graphs dates to Euler. The *faces* of an embedding are connected regions of the plane that are separated from each other by cycles of $G$. Euler showed that for any planar embedding, if $V$ is the set of vertices, $E$ the set of edges, $F$ the set of faces (regions of the plane that are connected in the embedding), and $C$ the set of connected components of the graph, then $|V| + |F| = |E| + |C| + 1$. Many other results about planar graphs can be proven using this formula. For instance, using the formula, it is easily proven with counting arguments that $K_5$ and $K_{3,3}$ are non-planar [12].

0-8493-8597-0/01/\$0.00+\$1.50
© 2001 by CRC Press, LLC

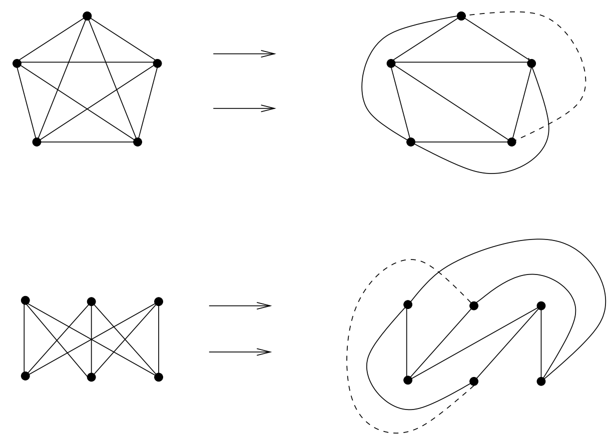


FIGURE 1.1: Two non-planar graphs. The first is the $K_5$, the complete graph on five vertices, and the second is the $K_{3,3}$, the complete bipartite class on two sets of three vertices each. Any attempt to embed them in the plane fails when a final edge cannot be inserted without crossing the boundary between two faces.

The famous 4-color theorem states that the vertices of a planar graph can always be partitioned into four independent sets; an equivalent statement is that a mapmaker never needs to use more than four colors to color countries on a map so that adjacent countries are of different colors. It remained open in the literature for almost 100 years and was finally proven with the aid of a computer program in 1976 [1, 2].

A *subdivision* of an edge $xy$ of a graph is obtained by creating a new node $z$, and replacing $xy$ with new edges $xz$ and $zy$. The inverse of this operation is the *contraction* of $z$, and only operates on vertices of degree 2. A subdivision of a graph is any graph that can be obtained from it by a sequence of subdivision operations. Since $K_5$ and $K_{3,3}$ are non-planar, it is obvious that subdivisions of these graphs are also non-planar. Therefore, a graph that has a subgraph that is a subdivision of $K_5$ or $K_{3,3}$ as a subgraph must be non-planar. Such a subgraph is said to be *homeomorphic* to a $K_{3,3}$ or a $K_5$.

A famous results in graph theory is the theorem of Kuratowski [21], which states that the absence of a subdivision of a $K_5$ or a $K_{3,3}$ is also sufficient for a graph to be planar. That is, a graph is planar if and only if it has no subgraph that is a subdivision of $K_{3,3}$ or $K_5$. Such a subdivision is known as a *Kuratowski subgraph.*

A *certifying algorithm* for a decision problem is one that produces an accompanying piece of evidence, or *certificate* that proves that its answer is correct. The certificate should be simple to check, or *authenticate.* Certifying algorithms are highly desirable in practice, where the possibility must be considered that an implementation has a bug and a simple yes or no answer cannot be entirely trusted unless it is accompanied by a certificate. The issue is discussed at length in [20]. Below, we describe a certifying algorithm for recognizing planar graphs. The algorithm produces either a planar embedding of the graph, proving that the graph is planar, or points out a Kuratowski subgraph, proving that it is not.

Next, let us consider a problem that is seemingly unrelated to that of finding a planar embedding of a graph, but which can be solved with similar data structures. Given a set $S$ of intervals of a line, let their *interval graph* be the graph that has one vertex for each of the intervals in $S$, and an edge between two vertices if their intervals intersect. That is, a graph is an interval graph if it is the *intersection graph* of a set of intervals on a line. Figure 1.2 gives an illustration.

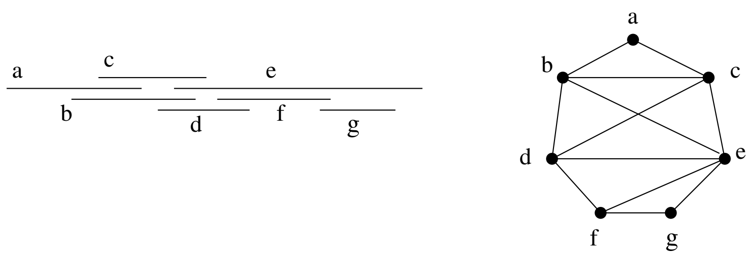


FIGURE 1.2: An interval graph is the intersection graph of a set of intervals on a line. There is one vertex for each of the intervals, and two vertices are adjacent if and only if the corresponding intervals intersect.

Interval graphs also come up in a variety of other applications, such as scheduling jobs that conflict if they must be carried out during overlapping time intervals. If an interval representation is given, otherwise NP-complete problems, such as finding a maximum independent set, can be solved in linear time [9].

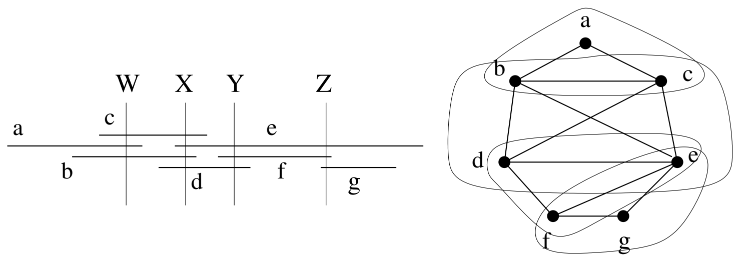
Given the intervals, it is trivial to construct their interval graph. However, we are interested in the inverse problem, where, given a graph $G$, one must find a set of intervals that have $G$ as their interval graph or else determine that $G$ is not an interval graph.

Interest in this problem began in the late 1950's when the noted biologist Seymour Benzer used them to establish that genetic information is stored inside a biological structure that has a linear topology [3]; this topology arises from the now-familiar structure of DNA. To do this, he developed methods of inducing mutations using X-ray photons, which could be assumed to reflect damage to a contiguous region, and for testing whether two of these mutations had common effects that indicated that the the damaged regions intersect. This gave rise naturally to a graph where each mutation is a vertex and where two vertices have an edge between them if they intersect. He got the result by showing that this graph is an interval graph.

Let us say that such a set $S$ is a *realizer* of the interval graph $G$ if $G$ is $S$'s interval graph. Benzer's result initiated considerable interest in efficient algorithms to finding realizers of interval graphs, since they give possible linear orderings of DNA fragments, or *clones*, given data about which fragments intersect [10, 5, 19, 26, 27, 28, 16, 11, 17]. A linear-time bound for the problem was first given by Booth and Lueker in [5]. Though the existence of forbidden subgraphs of interval graphs has long been well-known [22], the first linear-time certifying algorithm for recognizing interval graphs has only been given recently [20]; the certificate of acceptance is an interval realizer and the certificate of rejection is a forbidden subgraph.

When the ordering of intervals is unique except for trivial details, such as the lengths of the intervals and the relative placement of endpoints that intersect, this solves the *physical mapping problem* on DNA clones: it tells how the clones are arranged on the genome. Efficient algorithms for solving certain variations of this problem played a role in the assembling the genomes of organisms, and continue to play a significant role in genetic research [39]. For input data containing errors, Lu and Hsu [23] give an error-tolerant algorithm for the clone assembly problem.

A graph is a *circular-arc graph* if it is the intersection graph of a set of arcs on a circle. Booth conjectured that recognizing whether a graph is a circular-arc graph would turn out to be NP complete [4], but Tucker later found a polynomial-time algorithm [38]. McConnell has recently found a linear-time algorithm [29]. The problem of finding a certifying algorithm



$$\begin{array}{c|cccc} & W & X & Y & Z \\ a & 1 & 0 & 0 & 0 \\ g & 0 & 0 & 0 & 1 \\ c & 1 & 1 & 0 & 0 \\ e & 0 & 1 & 1 & 1 \\ d & 0 & 1 & 1 & 0 \\ f & 0 & 0 & 1 & 1 \\ b & 1 & 1 & 0 & 0 \end{array}$$

FIGURE 1.3: The clique matrix of a graph has one row for each vertex, one column for each maximal clique, and a 1 in row $i$ column $j$ iff vertex $i$ is contained in clique $j$. The maximal cliques of an interval graph correspond to points of maximal overlap in an interval representation. Ordering the columns of a clique matrix in the order in which they appear in an interval representation gives a consecutive-ones ordering of the clique matrix.

for the problem remains open.

Finding the maximal cliques of an arbitrary graph is hard: in fact it is NP complete to find whether a graph has a clique of a given size $k$. However, if a graph is *chordal* it is possible to list out its maximal cliques in linear time [32], and interval graphs are chordal. (A chordal graph is one that has no simple cycle on four or more vertices as an induced subgraph.) We may therefore create a *clique matrix*, which has one row for each vertex of the graph, one column for each maximal clique, and a 1 in row $i$, column $j$ iff clique $j$ contains vertex $i$.

***THEOREM 1.1*** *A chordal graph is an interval graph iff there is a way to order the columns of its clique matrix so that, in every row, the 1's are consecutive.*

To see this, suppose $G$ is an interval graph and $S$ is a realizer. Then, for each maximal clique $C$, a *clique point* on the line can be selected that intersects the intervals that correspond to elements of $C$ and no others. (See Figure 1.3.) Ordering the columns of the clique matrix according to the left-to-right order of the corresponding clique points ensures that the 1's in each row will be consecutive. Conversely, given a consecutive-ones ordering, the 1's in each row occupy an interval on the sequence of columns. It is easy to see that these intervals constitute a realizer of $G$, since two vertices are adjacent iff they are members of a common maximal clique.

Such an ordering of the columns of a 0-1 matrix is known as a *consecutive-ones ordering*, and a 0-1 matrix has the *consecutive-ones property* if there exists a consecutive-ones ordering of it. The main thrust of Booth and Lueker's algorithm consists of an algorithm for determining whether there exists a a consecutive-ones ordering of the columns of a 0-1 matrix. Their algorithm operates on a sparse representation of the matrix, and solves this in time linear in the number of 1's in the matrix. To test for the consecutive-ones property, they developed a representation, called a *PQ tree*, of *all* the consecutive-ones orderings of the columns. The tree consists of *P nodes* and *Q nodes*. The leaves of the tree are columns of the matrix, and the left-to-right leaf order of the tree gives a consecutive-ones ordering, just as it does when the order of children of a node are reversed, or when the order of children of a P node are permuted arbitrarily (see Figure 1.4). All consecutive-ones orderings of the columns can be obtained by a sequence of such rearrangements.

The PQ tree helps with keeping track of possible consecutive-ones orientations as they work by induction on the number of rows of the matrix. Each interval realizer of $G$ is given

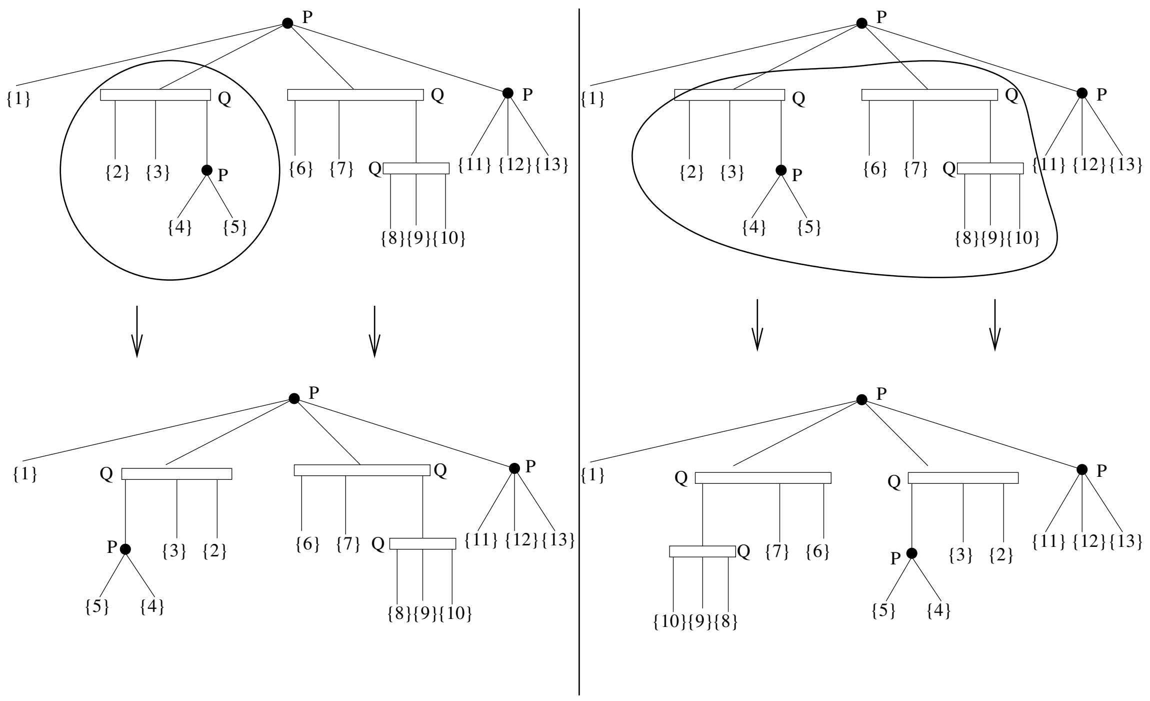


FIGURE 1.4: The leaves of a PQ tree are the columns of a consecutive-ones matrix. The left-to-right order of the leaves gives a consecutive-ones arrangement of the columns. So does the result of reversing the leaf descendants of a node. The order of leaves of a consecutive set of children of a P node can also be reversed to obtain a new consecutive-ones ordering. All consecutive-ones orderings can be obtained by a sequence of these reversals.

by a consecutive-ones ordering, except for minor details that do not affect the order of clique points.

The literature on problems related to PQ trees is quite extensive. Korte and Möhring [19] considered a modified PQ tree and a simpler incremental update of the tree. Klein and Reif [18] constructed efficient parallel algorithms for manipulating PQ trees. Hsu gave a simple test that is not based on PQ trees [15].

McConnell gives a generalization of the PQ tree to arbitrary 0-1 matrices, gives a linear-time algorithm for producing it, and a linear-time certifying algorithm for recognizing the consecutive-ones property [25].

The PQ tree play an important role in the linear-time algorithm of Lempel, Even, and Cederbaum for finding a planar embedding of planar graphs [24]. The algorithm takes advantage of the PQ tree's rich ability to represent families of linear orderings in order to keep track of possible arrangements of edges in an embedding of $G$.

Booth and Lueker's algorithm for constructing the PQ tree has a reputation for being difficult to understand and to program, and the many algorithms that have appeared since reflect an effort to address this concern. Their algorithm builds the tree by induction on the number of rows of the matrix. For each row, it must perform a second induction from the leaves toward the root. At each node encountered during this second induction, it uses one of nine *templates* for determining how the tree must change in the vicinity of the node. Recognizing which template must be used is quite challenging. Each template is actually a representative of a larger set of cases that must be dealt with explicitly by a program. These templates carry over into the use of the PQ tree in planar graph embedding.

The *PC tree* is an alternative introduced by Shih and Hsu [34] to address these difficulties. It is essentially the result of “unrooting” the PQ tree to obtain a free tree that awards no special status to any root node, and where notions of “up” and “down” in the tree have no meaning. This introduces a symmetry to the problem that is otherwise broken by the choice of the root, and once it is introduced, the various templates collapse down to a single case.

This suggests that the cases that must be considered in the templates are an artifact of an arbitrary choice of a root in the tree. The reason that this was not recognized earlier may have more to do with the fact that rooted trees are ubiquitous as data structures, whereas free trees are not commonly used as data structures. The need to root such a data may simply have been an assumption that people failed to scrutinize.

A matrix has the *circular-ones* property if the columns can be ordered so that, in every row, either the zeros are consecutive or the ones are consecutive. That is, it has the circular-ones property if the ones are consecutive when the matrix is wrapped around a vertical cylinder, which has the effect of eliminating any special status to any column, such as being the leftmost column.

Hsu [14] gives an algorithm using PC trees for solving the consecutive-ones problem. Hsu and McConnell [17] have shown that that both the PQ tree and the PC tree have remarkably simple definitions as mathematical objects. They are each precisely given by previously-known theorems on set families that had not previously been applied in this domain. Moreover, we show that the PC tree gives a representation of all circular-ones orderings of a matrix just as the PQ tree gives a representation of all consecutive-ones orderings.

Figure 1.5 illustrates how the PC tree represents the circular-ones orderings. The leaves are the columns of the matrix, and are arrayed around the large circle, which represents the circular ordering. The C nodes (double circles) have a cyclic order on their edges that can be reversed. We could think of them as coins with edges attached at discrete points around the sides, and that can be turned heads-up or tails-up, an operation that we will call a *flip*. The P nodes (black internal nodes) have no cyclic ordering. The circular-ones orderings of the columns of the matrix are just those that result from planar embeddings of this gadget that put the leaves on the outer circle. This description makes it obvious what family of circular orderings is represented: you can select an edge and reverse the order of all leaves that lie on one side of the edge, or you can reverse the order of a consecutive set of leaves if they are the leaves of a subset of the trees in the forest that would result from the removal of a P node.

Booth and Lueker showed that testing for the circular-ones property reduces in linear time to testing for the consecutive-ones property. It appears to be more natural to perform the reduction in the opposite direction. That is, to solve the consecutive-ones problem reduce it to the circular-ones problem, which can be solved with the PC tree instead of with the PQ tree. To do this, just add the zero vector as a new column of the matrix, compute the PC tree for the new matrix, and then pick it up by the leaf corresponding to the new column to root it. (See Figure 1.6.) In [17], it is shown that the subtree rooted at its child is the PQ tree for the original matrix.

## 1.2 The consecutive-ones problem

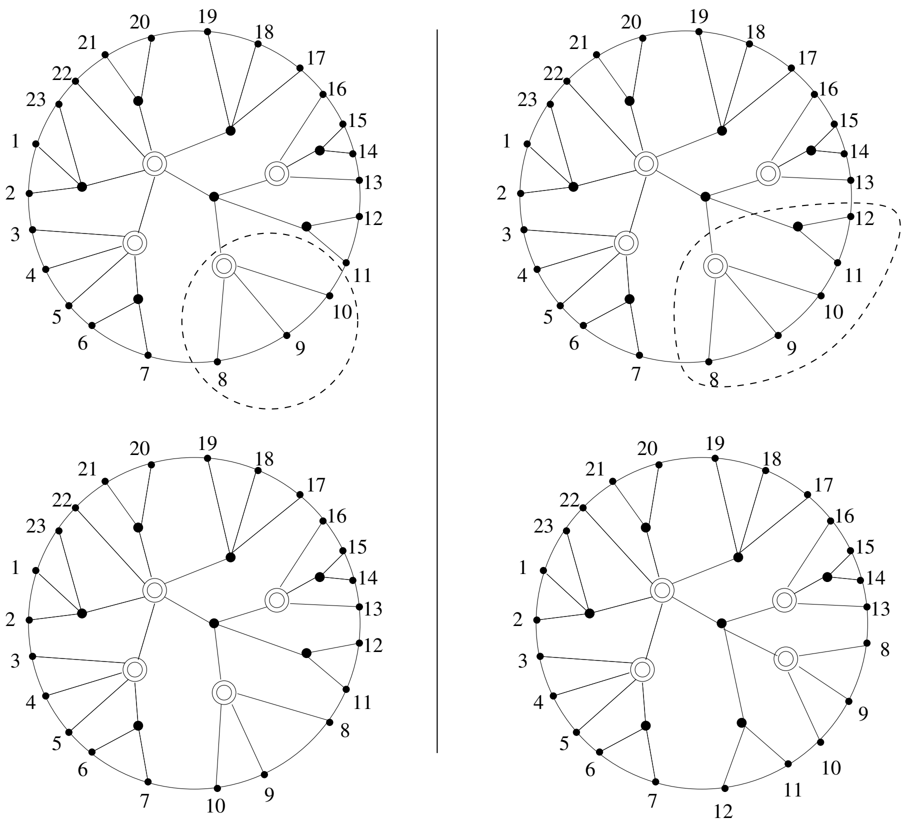


FIGURE 1.5: The PC tree can be viewed as a gadget for generating the circular-ones orderings of the columns. The C nodes are represented by double circles and the P nodes are represented by black dots. The subtree lying at one side of an edge can be flipped over to reverse the order of its leaves. The order of leaves of a consecutive set of subtrees that would result from the removal of a P node can also be reversed. All circular-ones orderings can be obtained by a sequence of such reversals.

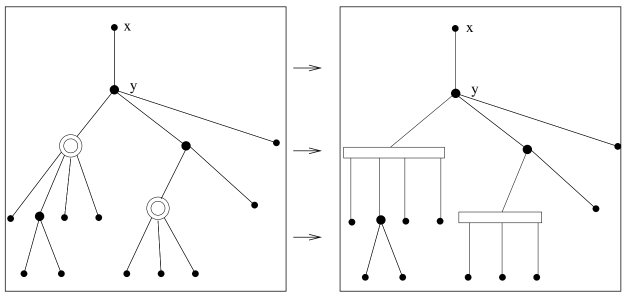


FIGURE 1.6: Assigning a new zero column $x$ to a matrix, computing the PC tree for it, and then picking the PC tree up at $x$ to root it, gives the PQ tree for the matrix, rooted at $y$, when the C nodes are reinterpreted as Q nodes.

In this section, we give an algorithm that is related to Booth and Lueker's algorithm, except that it uses the PC tree in place of the PQ tree.

Let is say that two subsets $X$ and $Y$ of a domain $V$ *strongly overlap* if $X \cap Y$, $X - Y \cup Y - X$, and $V - X - Y$ are all nonempty. We view the columns of a 0-1 matrix as a set $V$, and each row of the matrix as a subset of $V$ consisting of those columns where there is a 1 in the row.

A set $X$ is an *edge module* if it is the union of leaves in one of the two subtrees that results when an edge is removed. It is a *P module* if it is not an edge module but the union of leaves in a subset of the trees formed when a P node is removed. An edge or P module is an *unrooted module.* The key to understanding the construction of the PC tree is the fact that the unrooted modules are precisely those nonempty proper subsets of $V$ that do not strongly overlap any row of the matrix.

We construct the PC tree by induction on the number of rows of a matrix. The $i^{th}$ step of the algorithm modifies the PC tree so that it is correct for the submatrix consisting of the first $i$ rows of the matrix. As a base case, after the first step, the PC tree consists of two adjacent P nodes, with one of them adjacent to the leaves that correspond to ones in the first row and the other adjacent to the leaves that correspond to the zeros.

During the $i^{th}$ step, no new unrooted modules are created by adding a row, but some unrooted modules in the first $i-1$ rows may become defunct as unrooted modules once the $i^{th}$ row is considered. It is necessary to modify the tree so that it no longer represents these sets as unrooted modules.

Let the *full leaves* denote the leaves that correspond to ones in row $i$, and let the *empty leaves* denote those that correspond to zeros in row $i$. If an edge module $X$ becomes defunct in the $i^{th}$ step, then $X$ and $\overline{X}$ each contain both empty leaves and full leaves. Then $X$ corresponds to an edge whose removal separates the PC tree into two trees, each of which has both full and empty leaves. Let us call such an edge a *terminal edge.* Terminal edges must be removed from the tree, since they correspond to defunct edge modules. If $M$ has the circular ones property, these terminal edges form a path (See Figure 1.7). Let us call this path the *terminal path.* The *terminal nodes* are the nodes that lie at the ends of the terminal path. All nodes and edges that must be altered in Step $i$ lie on the terminal path. When there is a unique node of the PC tree that has both full and empty neighbors, we consider it to be a terminal path of length 0; this node assumes the role of both terminal nodes.

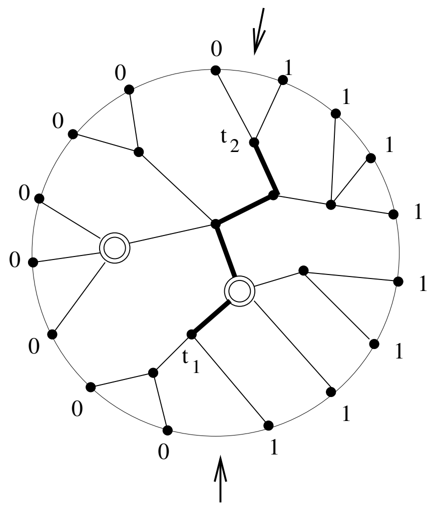


FIGURE 1.7: The edges that must be modified when a new row is added are those that represent two sets that have a mixture of zeros and ones, as these sets fail the criterion for being unrooted modules in the new row. If the matrix has the circular-ones property, these edges lie on a path, called the *terminal path*. The *terminal nodes* are the nodes $t_1$ and $t_2$ that lie at the ends of the terminal path.

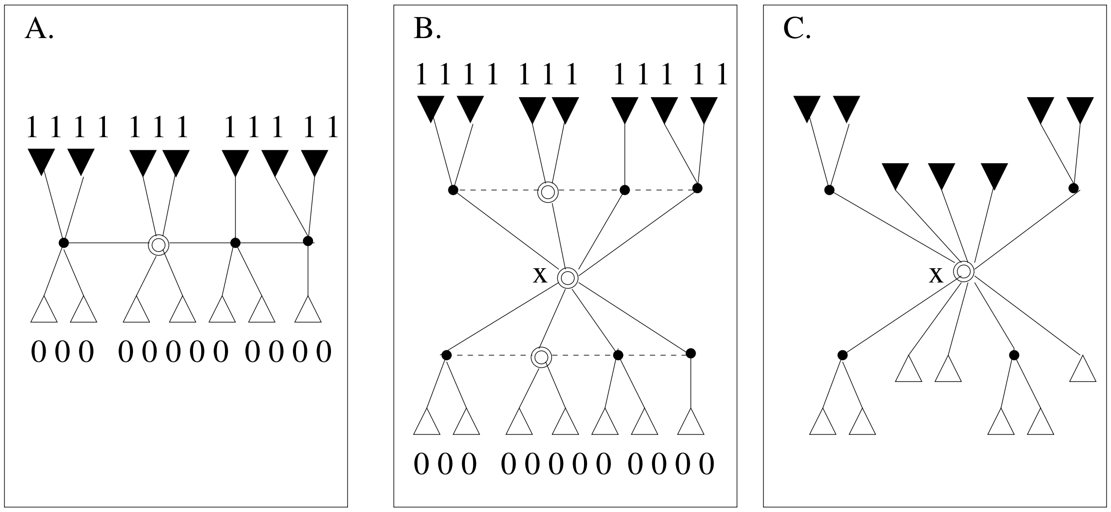


FIGURE 1.8: To update the PC tree when a new row is added to the matrix, flip the C nodes and order the P nodes on the terminal path so that the edges that go to trees whose leaves are zeros in the row lie on one side (white) and those that go to trees whose leaves are ones in the row lie on the other side (black). This is always possible if the new matrix has the circular-ones property (Figure A). Then divide each node on the terminal path into two parts, one that is adjacent to the black trees and one that is adjacent to the white trees (Figure B). Replace the edges of these two paths with a new C node, $x$, whose cyclic order reflects the order of nodes these two paths. Finally, contract each edge from $x$ to a C-node neighbor, and contract each internal node of degree two (Figure C).

**Algorithm 1.2** Constructing the PC Tree
*The initial PC tree is a P node that is adjacent to all leaves, which allows all* $(n-1)!$ *circular orderings.*
*At each row:*

- *Find the terminal path, and then perform flips of C nodes and modify the cyclic order of edges incident to P nodes so that all ones lie on one side of the path (see Figure 1.8.)*
- *Split each node on the path into two nodes, one adjacent to the edges to full leaves and one adjacent to the edges to empty leaves.*
- *Delete the edges of the path and replace them with a new C node* $x$ *whose cyclic order preserves the order of the nodes on this path.*
- *Contract all edges from* $x$ *to C-node neighbors, and any node that has only two neighbors.*

### 1.2.1 A Data Structure for Representing the PC Tree

For the implementation, we pair up the $n-1$ edges with $n-1$ of the nodes of the tree, so that each edge is paired with one of the nodes that it is incident to. This can be accomplished by rooting the tree at an arbitrary node in order to define a parent function, and then pairing each non-root node with its parent edge. It is worth noting that in contrast to the rooting

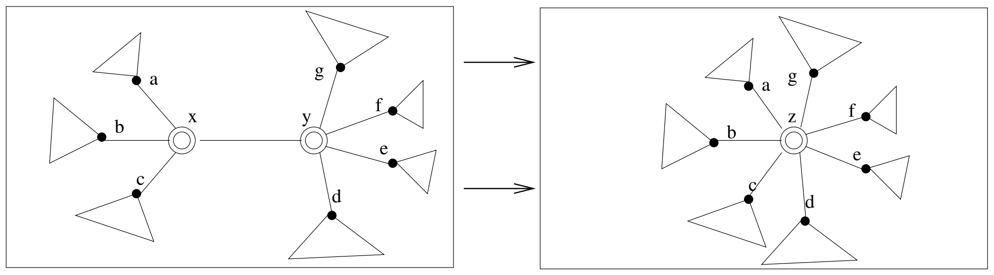


FIGURE 1.9: An order-preserving contraction of an edge $xy$. The neighbors of $x$ and $y$ are cyclically ordered. The edge is removed and $x$ and $y$ are identified, so that the cyclic order of neighbors of $x$ and $y$ about the edge is preserved.

of the PC tree, which serves to give a distinguished role to the root, the sole purpose of this is to make low-level operations more efficient. An example of such an operation is the problem of finding out whether two nodes of an unrooted tree are adjacent, which can be determined in $O(1)$ time if it is rooted, by examining the parent pointers of the two nodes.

An undirected graph is a special case of a directed graph where every arc $(u, v)$ has a *twin arc* $(v, u)$. Thus, we may speak of the directed *arcs* of the PC tree, not just its edges.

**DEFINITION 1.1** The data structure for the PC tree is the following. Each P node carries a pointer to the parent edge. Each edge $uv$ is implemented with two oppositely directed *twin arcs* $(u, v)$ and $(v, u)$. Each arc $(x, y)$ has a pointer to its two neighbors in the cyclic order about $y$, a pointer to its twin, and a *parent bit* label that indicates whether $y$ is the parent of $x$. In addition, if $y$ is a P node, then $(x, y)$ has a pointer to $y$. There is no explicit representation of a C node; its existence is implicit in the doubly-linked circular list of its incident edges that gives their cyclic order. No two C nodes are adjacent, so each of these edges has one end that identifies a neighbor of the C node, and another end that indicates that the end is incident to a C node, without identifying the C node.

If $(x, y)$ is an arc directed into $y$ and $y$ is a C node, then $y$ is not represented by an explicit record. We can find $y$'s parent edge by cycling through the the records for arcs that are directed into $y$ in either cyclic direction about $y$, until we reach an arc with the required parent bit. Thus, finding a C node's parent edge is not an $O(1)$ operation.

The data structure makes no distinction between the two directions in which a list can be traversed; this distinction is made only at the time when a traversal is begun. One must keep track of both the current and previous element. To move to the next element, one must retrieve both neighbors of the current element, and select the one that is different from the previous element.

Since we will deal with unrooted trees whose internal nodes can be cyclically ordered, it will be useful to define the cyclic order of edges incident to $z$ after the contraction. An *order-preserving contraction* is the one depicted in Figure 1.9, where the neighbors $x$ and $y$ are each consecutive and preserve their original adjacencies in the circular order of $z$'s neighbors.

Using this data structure, it takes $O(1)$ time to remove or insert a section of a list, given pointers to the endpoints. Since the list draws no distinction between forward and backward, a section of a list can be inserted in either order in $O(1)$ time.

It therefore supports an order-preserving contraction of an edge between two adjacent C nodes $x$ and $y$ in $O(1)$ time, given a pointer to $(x, y)$, in addition to allowing insertion or removal of an edge from a node's circular adjacency list or reversal of a section of a circular list in $O(1)$ time

### 1.2.2 Finding the Terminal Path Efficiently

Finding the terminal path is the only step where pointers to parent edges are required in the $i^{th}$ iteration.

Recall that we have defined full and empty leaves of the PC tree. For the internal nodes, let us say that a node is *full* if it is possible to root the tree so that all leaves in the subtree rooted at the node are full, and *empty* if it is possible to root the tree so that all leaves in its subtree are empty. Since each node has degree at least three, at most one of these designations applies at a node.

We use the following *full-partial labeling algorithm* to mark the full nodes. The efficiency of the algorithm is due to the fact that it avoids touching some of the empty nodes; it leaves them unmarked. We label a leaf as full if it corresponds to a column with a one in the $i^{th}$ row. We label an internal node as full if all of its neighbors except one have been labeled full. We label an internal node as *partial* if at least one of its neighbors has been labeled full. Whenever we label a node as full, we increment a counter in its non-full neighbor $x$ that records how many full neighbors $x$ has, labeling $x$ as full if this counter rises to one less than the degree of $x$. However, if $x$ is a C node, then since it is given implicitly by a circular list of neighbors, we do not keep an explicit counter at $x$. Nevertheless, it is easy to detect when all of its neighbors except one is full. Recall that no two C nodes are adjacent.

To perform the labeling in the presence of C nodes, we use an unrooted variant of the pointer borrowing strategy of [5]. We maintain *block-spanning pointers* from the first to last vertex and from the last to the first vertex in each consecutive block of full neighbors around the cycle that makes up $x$. Each time a new $y$ neighbor of $x$ becomes full, either $y$ becomes a one-element block, a block is extended by appending or prepending $y$, or two blocks and $y$ merged, by appending $y$ to one of the blocks and prepending it to the other. Each of these operations gives access in $O(1)$ time to the first or last vertex in each affected block, so it is trivial to update the block-spanning pointers in $O(1)$ time. A test of whether the first and last vertices in the resulting block share a non-full neighbor $z$ on the cycle takes $O(1)$ time. If $x$ passes this test, it is full, and the full-neighbor counter of $z$ is incremented.

Since every node of the PC tree has degree at least three, the number of full leaves is at least as great as the number of full internal nodes, and there are at most $k$ full leaves. Assigning full labels takes $O(k)$ time. The number of partial nodes is at most as great as the number of full nodes, since each full node has at most one partial neighbor. Assigning partial labels takes $O(k)$ time also.

Henceforth, let us call a node *full* or *partial* according to the final label assigned to it by the full-partial labeling algorithm.

The key insight for finding the terminal edges is the observation that an edge is terminal if and only if it lies on a path in the tree between two partial nodes.

In the special case where there are no terminal edges and the terminal path has length 0, it is the unique node that has both full and unmarked neighbors. It is easy to see that since there are no other such nodes, its unmarked neighbors are empty. This node is easy to find, given the marking of the nodes.

Otherwise, let the *apex* be the least common ancestor of the partial nodes. We find the terminal edges by starting at each partial node and extending a path up through its ancestors, marking edges on the path. We do this in parallel at all partial nodes, extending the paths at the same rate. When a path runs into another partial node, we stop extending that path. If a path extends above the apex, we may or may not detect this right away. Eventually, we will be extending only one path, at which point, the marked edges form a connected subtree. The apex is the first point below the highest point of this subtree that is either partial or has two paths entering it. Unmarking the edges from the marked edge down to the apex leaves edges marked iff they are terminal edges.

If $x$ is a node on the terminal path other than the apex, then the parent of $x$ is also on the terminal path. If $x$ is a P node, this takes $O(1)$ time, since it has a pointer to its parent.

If $x$ is a partial C node that has a full neighbor, we may assume that we have a pointer to the edge to this neighbor, since this is provided by the full-partial labeling algorithm. This is always the case at a terminal node, which has a full neighbor and an empty neighbor. As we climb the terminal path toward the apex, when reaching a C node $y$ from its child $x$ on the terminal path, we obtain a pointer to the edge $(x, y)$, since $x$ is a P node and $(x, y)$ is its parent edge. We obtain pointer even if $y$ has no full neighbor.

The key to bounding the cost of finding $x$'s parent when $x$ is a C node on the terminal path is the observation that if it has any full neighbors, the full neighbors are consecutive, and the edges to its neighbors on the terminal path are adjacent to the full neighbors in the cyclic order. Thus, if it has no full neighbor, we can look at the two neighbors of the child edge in the cyclic order, and one of them must be the parent edge. This takes $O(1)$ time. If $x$ has a full neighbor, then we can cycle clockwise and counterclockwise through the edges to full neighbors. Of the first edges clockwise and counter-clockwise that are not to full neighbors, one of these is the parent. In this case, the cost of finding the parent is proportional to the number of full neighbors $x$.

If $x$ does not lie on the terminal path, then this procedure may fail, in which case we detect that it is not on the terminal path, or it may succeed, in which case we can bound the cost of finding the parent in the same way.

If the union of all paths traversed has $p'$ nodes and there are $k$ ones in row $i$, then the total cost is $O(p' + k)$. However, the number of nodes in these paths that are not on the terminal path is at most the number of nodes that are on the terminal path, because of the way the paths are extended in parallel. The following summarizes these observations.

**LEMMA 1.1** If the terminal edges form a path and the full and empty neighbors can be flipped to opposite sides of it, then finding the terminal path in the $i^{th}$ step takes $O(p + k)$ time, where $p$ is the length of the terminal path and $k$ is the number of full nodes.

If the conditions of the lemma are not satisfied, the matrix does not have the circular-ones property, and is rejected.

### 1.2.3 Performing the update step on the terminal path efficiently

The number of full neighbors of nodes on the terminal path is bounded by the number $k$ of ones in the $i^{th}$ row. Before splitting a node, we record its neighbors on the terminal path, and then delete the edges to these neighbors, in $O(1)$ time. We then split the node by splicing out the full neighbors, and forming a new node with them. The remainder of the old node serves as the other half of the split. This takes time proportional to the number of full neighbors. Since the number of full neighbors of nodes on the terminal path is bounded by the number of ones in the $i^{th}$ row, the total time for this is $O(p + k)$. Creating the new C node $x$ and installing edges to the split nodes in the required cyclic order then takes $O(p)$ time.

Since our data structure includes a parent function, we must assign the parent bits to the new edges. Let $y$ be the copy of the apex that retains its parent edge after the split of the apex. Except in the case of the apex, the parent edge of every vertex on the terminal path is an edge of the terminal path, and these edges have been deleted. Thus, none of these nodes have a parent edge. We make $x$ the new parent of these nodes, and we let $y$ be the parent of $x$. This takes $O(p)$ time by setting the appropriate bits in the $O(p)$ edges incident to $x$.

The operation of deleting a C node $z$ from $x$'s neighborhood and replacing it with the neighbors of $z$ is just a contraction of the edge $xz$, depicted in Figure 1.9, which takes $O(1)$ time. These observations can be summarized as follows:

**LEMMA 1.2** Updating the tree during the $i^{th}$ step takes $O(p + k)$ time, where $p$ is the length of the terminal path and $k$ is the number of full nodes.

### 1.2.4 The linear time bound

Assume that the matrix is given in sparse form, where, for each row, the set of columns where the row has a one is listed. Let us assume that every row and every column has at least one nonzero entry, since it can otherwise be eliminated in a preprocessing step. A linear time bound is one that is proportional to the number of nonzero entries in the matrix.

The algorithm processes one row at a time of the matrix $M$. Let $T$ denote the current state of the PC tree after the first $i$ rows have been processed. That is, $T$ is the PC tree for the submatrix induced by the first $i$ rows. Let $C_i$ be the set of C nodes and let $P_i$ be the set of $P_i$ nodes in $T$, and let $u_i$ be the number of ones in the rows of the matrix that have not yet been processed. If $x$ is a node of $T$, let $deg(x)$ denote the *degree*, or number of neighbors of $x$ in $T$.

If, when processing a row, we could update the PC tree in time proportional to the number of ones in the row, the linear time bound would be immediate. Unfortunately, the update step does not conform to this bound. Therefore, we use a technique called *amortized analysis* [9] , which uses an accounting scheme whereby updates that exceed this bound borrow credits from updates that were completed with time to spare. The analysis shows that, even though there is variability in the time required by the updates, the aggregate cost of all updates is linear.

We adopt a budget where we keep an account that must have a number of credits $\phi(M, i)$ in reserve, where $\phi(N, i)$ is given by the following:

- $\phi(M, i) = 2u_i + |C_i| + \sum_{x \in P_i}(deg(x) - 1)$.

Such a function is often called a *potential function.* Each credit can pay an $O(1)$ operation. Any operation that reduces $\phi$ allows us to withdraw credits from the account to pay for some of the operations. Since $\phi(M, 0)$ is $\Theta(m)$, we must pre-pay the cost of later operations by making a deposit of $\Theta(m)$ credits to the account. If we can maintain this reserve as we progress and still pay for all operations with withdrawals, then, since $\phi$ never runs the account to zero, the running time of the algorithm is $O(m)$.

To cover the costs in row $i$, we must be able to withdraw $k + p$ credits, by Lemmas 1.1 and 1.2, where $k$ is the number of full nodes. Since every full node has at least two full neighbors, the number of marked nodes is at most two times the number of 1's in row $i$, so $k$ credits are freed up by the decrease from $2u_i$ to $2u_{i+1}$.

Each C node on the terminal path is first split, but then both of these copies are contracted. This decreases the number of C nodes by one without changing the degrees of any P nodes, so spending a credit for each C node on the path is within the budget. Suppose a

P node does not lie at the end of the terminal path. If it is split, the sum of degrees of the two parts is the same as the degree of the original, after the terminal-path edges are deleted and replaced with an edge to the new C node. However, in calculating $\phi$, subtracting one from the degrees of two P nodes instead of one frees a credit. If the P node has only empty neighbors or only full neighbors, it is not split. In this case its degree decreases by one when its two incident terminal-path edges are deleted and replaced by a single edge to the new C node. A P node at the end of the terminal path fails to free up a credit, but there are only $O(1)$ of these. Contractions of nodes of degree two free up a credit, whether they are C nodes or $P$ nodes. $\Theta(k + p)$ credits for row $i$ can be paid out while adhering to the budget.

## 1.3 Planar Graphs

In this section, we describe an algorithm due to Shih and Hsu [34] that uses PC trees to recognize whether a graph $H$ is planar. If it is, the algorithm returns a planar embedding, and if it is not, it points out a subgraph that is a subdivision of a $K_{3,3}$ or a $K_5$.

Linear time planarity test was first established by Hopcroft and Tarjan [13] based on a *path addition approach*, which finds a path in the graph and uses it to break the problem down recursively. A *vertex addition approach*, originally developed by Lempel, Even and Cederbaum [24], was later improved by Booth and Lueker [5] (hereafter, referred to as the *B&L algorithm*) to run in linear time using PQ trees. This approach adds one vertex at a time, updating the PQ tree to keep track of possible embeddings of the subgraph induced by vertices added so far. Both of these approaches are quite complex. Furthermore, both approaches use separate algorithms for recognition and embedding (Chiba et al [8]). Several other approaches have also been developed for simplifying the planarity test (see for example [7, 31, 35, 36]) and the embedding algorithm [6, 30]. Shih and Hsu [33] developed a linear time test, which has been referred to as the simplest linear time planarity test by Thomas in his lecture notes [37]. Independently, Boyer and Myrvold discovered a similar algorithm to the PC tree approach [6]. Later, in [34], they implemented the algorithm based on PC trees, which will be referred to as the *S&H algorithm.* When the given graph is not planar, the algorithm immediately produces explicit Kuratowski subgraphs. Furthermore, the recognition and embedding are done simultaneously in the algorithm.

### 1.3.1 Preliminaries

To represent a planar embedding, it suffices to find, for each vertex, the clockwise circular ordering induced by the planar embedding on its incident edges. Given these circular orderings, there are algorithms that can assign spatial coordinates to the nodes. Here, we deal only with the problem of finding these circular orderings, and refer to them collectively as the *embedding* of the graph.

As the algorithm progresses, more of the embedding becomes known. In particular, S&H comes to know the cyclic order of edges incident to a subset of the vertices. When the cyclic order of a vertex is known, it does not know whether this order should be clockwise or counterclockwise in the embedding, so it uses a C node to represent its cyclic order. Collectively, the C nodes represent the partial embedding known so far. Let us call such a graph with C nodes a *constrained graph.* The data structure for implementing the C node is the same as the one given in Definition 1.1; the algorithm continues to ensure that no two C nodes are adjacent.

If $T$ is a depth-first spanning tree (DFS tree) of an undirected graph $G$, all edges of $G$

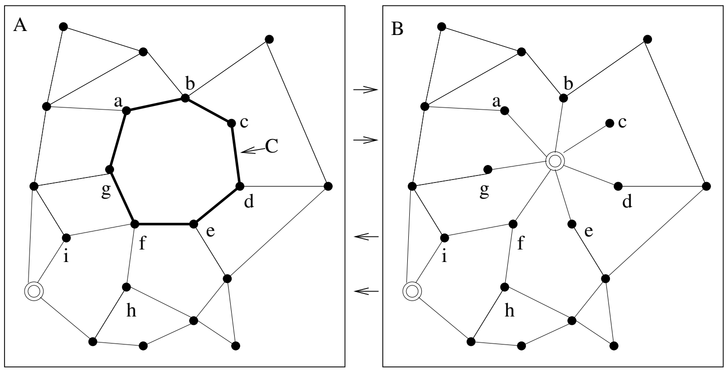


FIGURE 1.10: A *cycle replacement* consists of selecting a cycle in a graph, adding a new C node whose neighbors ordering gives the ordering of nodes on the cycle, and deleting the edges of the cycle. As we illustrate in parts C and D of Figure 1.11, below, it is applied to a graph that arises partway into the induction step. It's inverse operation is a `C-node replacement`.

are *tree edges* (edges of $T$) or *back edges* (edges between a descendant and an ancestor in $T$ [9]. A vertex is a *back vertex* if it has an incident back edge from one of its descendants. Since there are $n-1$ edges in $T$, the number of back edges is $m-n+1$. We may therefore refer to the number of back edges without reference to any particular DFS tree.

A *cut set* is a set of vertices of a connected graph whose removal from the graph disconnects it. An *articulation vertex* in a graph is a vertex whose deletion disconnects the graph. A graph is *biconnected* if it has no articulation vertex [9]. A *biconnected component* of a graph is a maximal biconnected subgraph. The articulation vertices can be found in linear time [9], so it suffices to embed each biconnected component separately, and then connect them by their adjoining articulation vertices. Henceforth, therefore, we may assume that $G$ is biconnected.

Given a planar embedding of a constrained graph $G$, and a cycle $C$ in $G$ that is a cut set, let us say that $C$'s *subembedding* is $C$ and everything internal to it in the embedding.

A *C-node replacement* is the following operation (See Figure 1.10): If $(y_1, y_2, ..., y_k, y_1)$ is the cyclic order of neighbors of $x$, install an edge $y_i y_{(i+1) mod k}$ for each $i$ from 1 to $k$, then delete $x$. This replaces $x$ with the cycle $(y_1, y_2, ..., y_k, y_1)$. A *cycle replacement* is the inverse of this operation: select a cycle $C$, and insert a new C node $x$ whose cyclic ordering of neighbors gives the vertices of $C$ in order, and then delete the edges of $C$.

### 1.3.2 The Strategy

The algorithm of S&H can be described as a recursive algorithm, `Embed`. The graph passed to the initial call has no C nodes, but graphs passed to lower calls will have them. A DFS spanning tree is also passed to the call, and since the DFS tree may contain C nodes, it is a PC tree.

For now, assume that the initial graph is planar; later, we show how to modify the algorithm so that it returns a Kuratowski subgraph when this is not the case. The input graph to each recursive call is also biconnected. `Embed` works by induction on the number $m-n+1$ of back edges (see Figure 1.11):

1. Choose a cycle $C$ that is a cut set and return its subembedding $A$ in some

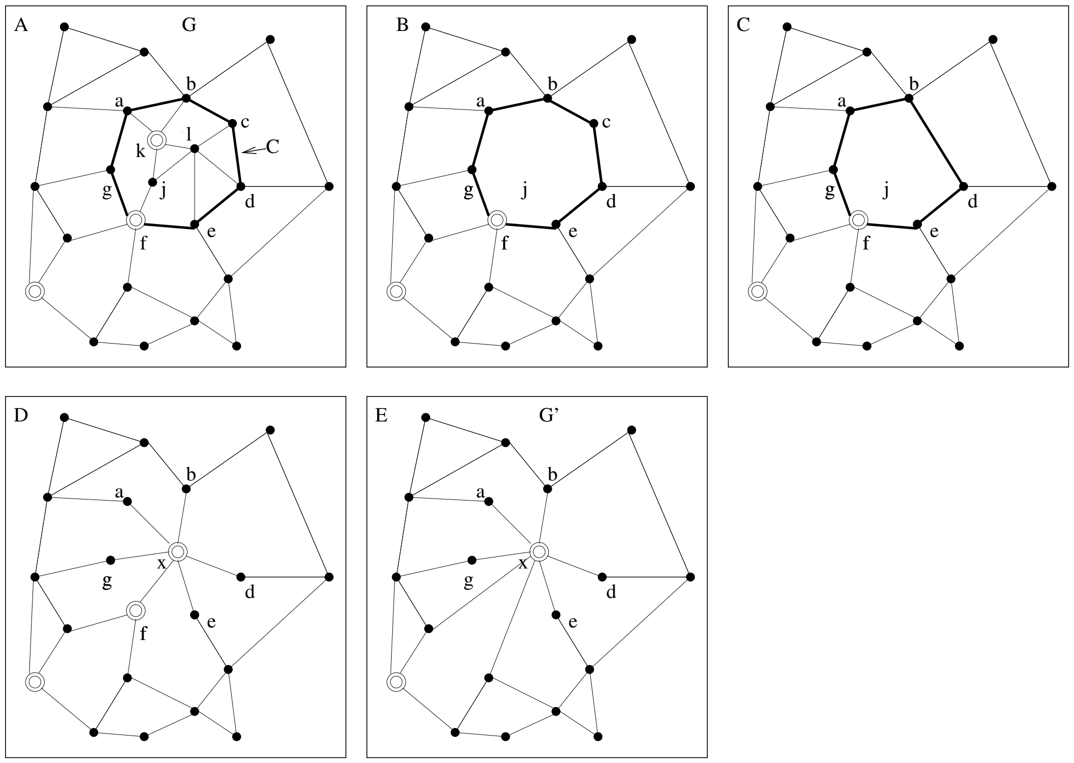


FIGURE 1.11: The `Embed` operation. `Embed` finds a cycle C and its subembedding $A$ in an unknown planar embedding of $G$. `Embed` removes elements of $A$ that are internal to $C$, contracting resulting vertices of degree 2 on $C$, and performs a cycle replacement on $C$, inserting a new C node $x$. It then performs contractions to eliminate C-node neighbors of $x$. The result is the graph $G'$. By induction on $m - n + 1$, a recursive call can be used to find an embedding of $G'$. Performing a C-node replacement of $x$ gives a planar embedding where $C$ is a face; inverting the foregoing operations inserts the planar embedding of $A$ inside of it.

unknown planar embedding of $G$ (Figure 1.11, part A).

2. Remove elements internal to $A$ to obtain graph $G_2$, which has a planar embedding where $C$ is a face (Figure 1.11, part B).
3. Contract nodes on $C$ that now have degree 2 to obtain a cycle $C'$ (Figure 1.11, part C).
4. Perform a cycle replacement on $C'$ to obtain a constrained graph $G_3$ (Figure 1.11, part D).
5. Perform edge contractions between adjacent C nodes in $G_3$ to obtain a a biconnected constrained graph $G'$ where no two C nodes are adjacent (Figure 1.11, part E).
6. By induction on the number $m-n+1$ of back edges, we may call `Embed` recursively on $G'$ to obtain a planar embedding of it.
7. On the planar embedding of $G'$, perform the inverses of steps 6, 5, 4 and 3 to obtain a planar embedding of $G$.

The reason for contracting nodes on C of degree two in Step 3 is to ensure that, like $G$, $G'$ is biconnected. Failure to do so would result in pendant nodes, such as node $c$ in Figure 1.10.

Because of the way $C$ is selected, the base case will be a biconnected constrained graph $G$ with a vertex $n$ such that $G - n$ is a PC tree. $G - n$ is trivial to embed, and since this embedding has only one face, it is trivial to add $n$ and its incident edges to this embedding to obtain an embedding of $G$.

## 1.3.3 Implementing the recursive step

Let us name the vertices according to their postorder numbering on a DFS tree $T$ of $G$. If $i$ is a vertex, we let $T_i$ denote the subtree of $T$ rooted at $i$. Let us say that a vertex $j$ is *earlier* than vertex $i$ if $j < i$.

The following are the inputs to `Embed`.

**I1:** A biconnected constrained graph $G$.

**I2:** The earliest back vertex $i$ in $T$;

**I3:** A DFS tree $T$ of $G$ where all C nodes in the tree are earlier than $i$. The DFS tree is implemented as in Definition 1.1, except that the circular lists of edges incident to a C node can include non-tree edges. The parent bits of Definition 1.1 are required only on tree edges, and are consistent with the rooting of the DFS tree.

**I4:** An ordered list of $i$ and all later vertices, ordered in postorder on the DFS tree.

### The Terminal Path

Let $i$ be the earliest back vertex, let $r$ be a child of $i$ in the DFS tree whose subtree $T_r$ in the DFS tree has a back edge to $i$. Since $r$ is earlier than $i$, $T_r$ is an induced subgraph of the constrained graph $G$ that has no back edges, hence it is a tree.

A trivial case occurs when $i$ is the root $n$ of the DFS tree. Since $G$ is biconnected, $n$ is not an articulation vertex, so $T_r$ is unique, and $T_r = G - n$. $T_r$ is trivial to embed, and since this embedding has only one face, it is also trivial to add $n$ and its incident edges to this embedding. This is the base case referred to in Section 1.3.2.

Otherwise, $i < n$. For ease of presentation, let us imagine, but not explicitly create, an *unrooted* tree $T'_r$ as follows. For each edge $(x, j)$ from a node $x$ of $T_r$ to a node $j \geq i$, add an edge $(x, j_x)$ to $T_r$. Note that this applies to the tree edge $(r, i)$, yielding $(r, i_r)$. The result is a PC tree, but there may be multiple copies of each back vertex $j$, one for each edge from a node of $T_r$ to $j$.

By analogy to Section 1.2.2, let us consider a leaf $j_x$ of $T'_r$ to be *full* if $j = i$ and *empty* otherwise. Since $G$ is biconnected, every leaf of $T_r$ has a neighbor greater than or equal to $i$; otherwise, its neighbor in $T_r$ would be an articulation vertex. Therefore, every node of $T_r$ is an internal node in $T'_r$. In addition, since $i < n$ and $i$ is not an articulation vertex, $T_r$ has edges both to $i$ and to proper ancestors of $i$, so $T'_r$ has both empty and full leaves. Finally, since $i$ has an incident tree edge and an incident back edge from $T_r$, $T'_r$ has at least two full leaves.

As in Section 1.2.2, let us consider an internal node $x$ to be *full* if there is a rooting of $T'_r$ where $x$'s subtree only has full leaves, and *empty* if there exists a rooting where its subtree only has empty leaves. It is a *terminal edge* if neither of its endpoints is full or empty. If the terminal edges form a path, this is the *terminal path.* If there are terminal edges but they do not form a path, then $T'_r$ has no terminal path. As before, if there are no terminal edges, the terminal path has length 0 and consists of a single node.

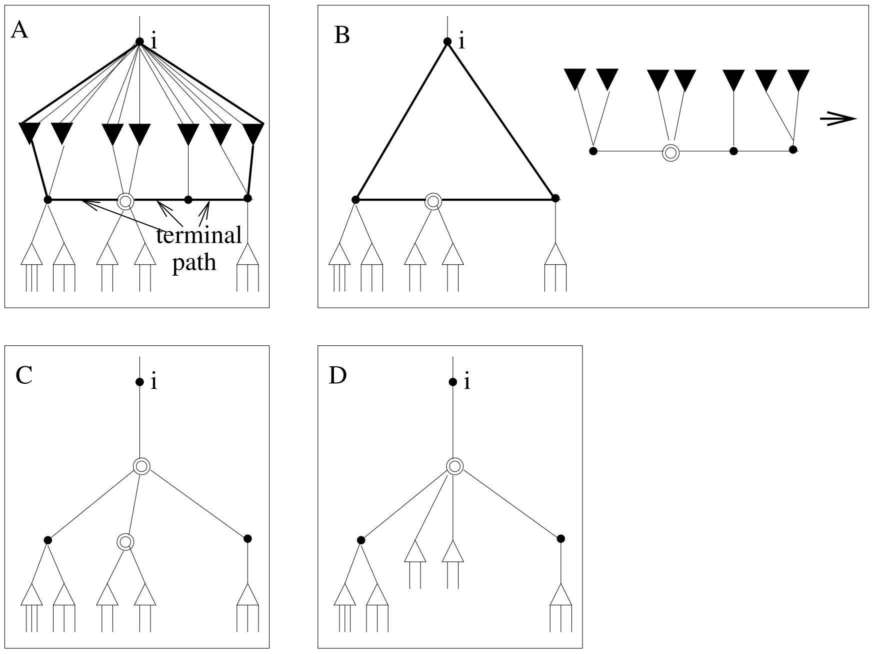


FIGURE 1.12: The induction step of `Embed`. The cycle C selected by `Embed` consists of $i$, the terminal path, and the leftmost and rightmost paths to $i$ from the terminal nodes after full and empty subtrees have been flipped to opposite sides of the terminal path (A). The full nodes and their back edges are trivial to embed inside $C$, since all of their back edges go to $i$. The nodes internal to $C$ are to be removed. If they are removed, however, the nodes on the paths from $i$ to the terminal nodes will have degree 2 so they can contracted out of the cycle. The net effect is to remove all full nodes from $G$, and leave a cycle consisting of $i$ and the terminal path. This is accomplished by splitting the terminal path, as in the consecutive-ones problem, removing the full side of the split, and inserting an edge from $i$ to each terminal node (B). A cycle replacement is performed on this cycle (C), and, as in the consecutive-ones problem, an order-preserving contraction is performed to remove C-node neighbors of of the new C node (D). This yields $G'$; performing a recursive call on $G'$ and inverting the steps from $A$ to $D$ on the resulting embedding gives a planar embedding of $G$.

We give a proof of the following in Section 1.3.5:

**LEMMA 1.3** If the constrained graph $G$ has a planar embedding, it has a terminal path, and nodes on this path can be flipped so that all full subtrees lie on one side and all empty subtrees lie on the other, without violating the cyclic order of any C node.

This gives a planar embedding of $T_r'$ in which all copies of $i$ can be joined without crossing any back edges, as illustrated in Figure 1.12 (A). By the definition of $T_r$, there must be a back edge to $i$, and since $G$ is biconnected, there must be a back edge from $T_r$ to a proper ancestor of $i$; otherwise $i$ would be an articulation vertex, since we are in an induction step where $i \neq n$. The full subtrees, the terminal path, and $i$ form an induced subgraph with a bounding cycle in this embedding. This bounding cycle is the cycle $C$ referred to in the overview of the algorithm in Section 1.3.2.

As described in the overview, only the cycle, minus its nodes of degree two, is retained for the recursive call (Figure 1.12, part B), and it is replaced by a C node (Figure 1.12, part C). This results in adjacent C nodes whenever there is a C node on the terminal path, which is remedied with an $O(1)$-time contraction is performed on them, as depicted in Figure 1.9, with a result illustrated in Figure 1.12, part D.

A recursive call on the resulting graph provides an embedding of it. Inverting the operations depicted in parts D through B of Figure 1.12 yields a face into which the already-known embedding of the full subtrees can be inserted to yield a planar embedding of the original constrained graph.

### Finding the Terminal Path

We use the full-partial labeling algorithm of Section 1.2.2 to label the full internal nodes of $T'_r$. This does not require creating $T'_r$ explicitly, since $T_r$ and its back edges represent $T'_r$ implicitly, and the full-partial labeling algorithm can be run on this representation by considering $i$ to be full and its ancestors to be empty.

However, we must confront an annoying detail that we didn't have in Section 1.2.2, which is that internal nodes can have degree 2. This allows the possibility that an internal node can be both empty and full. This can happen as follows. Let $x$ be a full node, let $y$ be an empty neighbor of degree 2, and let $z$ be $y$'s other neighbor. Rooting $T'_r$ at $z$ gives $y$ a subtree whose leaves are all full, and rooting it at $x$ gives $y$ a subtree whose leaves are all empty. Therefore, $y$ is *ambiguous*.

If it is run without modification, the full-partial labeling algorithm of Section 1.2.2 will label ambiguous nodes as full. However, we can detect the first time it labels an ambiguous node $y$. In this case, $x$ is a full neighbor that has just notified $y$ that it is full. If $x$ has degree 2, then it is also ambiguous, contradicting our choice of $y$. If it has degree 1, it is the only full leaf, contradicting the fact that $T'_r$ has at least two full leaves. Therefore, $x$ has degree greater than two, and full leaves can be reached from $y$ only by going through $x$.

This situation is easily detected: when it is time for $x$ to notify $y$ that it has become full, $x$ is the only node that has been labeled full but not yet notified its non-full neighbor, and $y$ has degree 2. In this case, we halt the full-partial labeling algorithm early, and select $x$ to be the only node in a terminal path of length 0. Aside from this detail, the full-label algorithm is the same as in Section 1.2.2.

By the analysis of Lemma 1.1, the running time is proportional to the number of terminal edges plus the number of full nodes if the conditions of Lemma 1.3 are met. If they are not met, the graph can be rejected. However, in this case, we still want to know the terminal edges since we will use them to produce a Kuratowski subgraph. The algorithm for finding them is easily implemented to run in time proportional to the size of $G$ in this case, as it requires finding only the subtree of edges that lie on paths between partial nodes. They can be labeled by rooting the tree at one of the partial nodes and working upward from the other partial nodes.

### The linear time bound

The analysis of the complexity of the full-partial algorithm of Section 1.2.2 made use of the fact that there are no nodes of degree two. We have not ensured that this is true of $T'_r$. The main problem that this causes is that the number of full nodes is not asymptotically bounded by the number of full leaves, so we can no longer claim that the running time of the full-partial labeling algorithm is bounded by the number of full leaves.

Instead, S&H makes use of the observation that all full nodes are deleted from the recursive call. We may therefore use a potential function that charges the costs of the full-partial labeling algorithm to nodes and edges that are removed from the recursive call, at $O(1)$ cost per node or edge.

The potential function is similar to the one for the consecutive-ones property, except that $u_i$ denotes the number of nodes and edges of the graph:

$$\phi(M, i) = 2u_i + |C_i| + \sum_{x \in P_i} (deg(x) - 1)$$

We show that the value of the potential function for the recursive call drops by an amount proportional to the set of operations performed outside of the recursive call, such as running the full-partial labeling algorithm.

The analysis of the cost of the $O(p)$ operations is unaffected. Since the full nodes are deleted, the $\Omega(k)$ drop in the potential function pays for the remaining $O(k)$ operations. The remaining operations of finding the terminal path are analyzed as in the consecutive-ones problem.

Finding and splitting the terminal path is performed just as it is in the consecutive-ones problem. By Lemmas 1.1 and 1.2, this takes time proportional to the number $k$ of full nodes plus the length $p$ of the terminal path, and covers the cost of removing the nodes internal to $C$ illustrated in Figure 1.12 (B).

We must now analyze the cost of meeting the preconditions I1-I4 in each recursive call. I1 through I4 can easily be met in $O(n + m)$ time for the initial call, where all nodes are P nodes, using standard techniques from [9].

Given I1 - I4 for $G$, we describe how to modify them so that they can be met in the recursive call on $G'$.

**LEMMA 1.4** A rooted spanning tree of an undirected graph is a DFS tree if and only if all non-tree edges are back edges.

The necessity of this condition is common knowledge [9]. For the sufficiency, observe that if $T$ is a rooted spanning tree such that every non-tree edge is a back edge, then ordering the adjacency lists so that tree edges appear first and calling DFS at the root of $T$ will force it to adhere to $T$ as the DFS tree.

For Input I4, let $T'$ be the DFS tree passed to the recursive call. Since all differences between $T$ and $T'$ occur in subtree $T_i$, the postorder of elements later than $i$ in $T$ is also the same as in $T'$. The input is met by searching forward in the preorder list for the next back vertex, and discarding the traversed elements from the front of the list.

For Input I3, let us make the new C node $x$ inserted inside $C$ be a child of $i$, hence a parent of its other neighbors in the DFS tree that is passed to the recursive calls. This requires us to label the parent-bit labels of edges incident to the new C node before contractions, as in Figure 1.12 (C). We have already bounded the cost of touching these edges, so this does not affect the asymptotic running time. Any remaining C nodes that are removed by $O(1)$-time contractions, as in Figure 1.12 (D) have the contracted edge as their parent edge, so the new node becomes the parent of their empty subtrees, without requiring any further relabeling of parent bits.

Since all back edges in $T$ go from descendants to ancestors, it is easy to see that this is also true of $T'$. $T'$ is a depth-first spanning tree of $G'$, by Lemma 1.4, so the conditions of I3 are satisfied by $T'$.

For Input I1, Step 3 of the description of Section 1.3.2 ensures that every node on the cycle has an incident edge to an empty node in $T'_r$. Therefore, the remaining graph is biconnected, since for every node in $T_r$, there are two paths to a proper ancestor of $i$: one of them by traveling upward on tree edges, and one of them by traveling downward through an empty tree and then to the ancestor on a back edge.

For Input I2, if $i$ ceases to be a back vertex in $G'$, then vertices are examined in ascending order in the list I4, starting at $i$ until the next back vertex, $i'$, is encountered. Since these searches are monotonically increasing, the total costs of them over all recursive calls is linear.

### 1.3.4 Differences between the original PQ-Tree and the new PC-Tree Approaches

Whenever a biconnected subgraph is created, the algorithm uses a subset of vertices in its bounding cycle as representatives to be used for future embedding. The embedding of each biconnected component is temporarily stored so that, at base case, when the graph is declared planar, a final embedding can be constructed by tracing back and pasting the internal embedding of each biconnected component inside its bounding cycle.

The way S&H adopted PC trees in their planarity test is entirely different from that of B&L's application of PQ trees in Lempel, Even and Cederbaum's planarity test [24]. 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 S&H's approach, a P node is also a vertex of the original graph $H$, a C node denotes a biconnected subgraph, and nodes adjacent to the new node can be scattered anywhere, both as internal nodes and as leaves in the PC tree. Thus, S&H's 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 Lempel, Even, and Cederbaum's approach, there is 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 an arbitrary graph [13].

Other aspects of handling the PC tree are adapted from B&L's approach, such as the handling of of Q nodes (or C nodes) during execution of the full-partial labeling algorithm.

### 1.3.5 Returning a Kuratowski Subgraph when $G$ is Non-Planar

In this section, let $H$ denote the unconstrained graph that is passed into the initial call, let $G$ denote the constrained graph that is passed into a lower recursive call, and let $G'$ denote the constrained graph that is passed into the next recursive call down from $G$.

The graph $H$ passed in at the highest-level call has no C nodes. In any recursive call on a constrained graph $G$, all P nodes are vertices of $H$. Therefore, all neighbors of a C node are vertices of $H$.

**LEMMA 1.5** A unique cycle $C$ of $H$ that has the following properties can be constructed from a C node $x$ of $G$. Let $(x_1, x_2, ..., x_k)$ be the cyclic order of $x$'s neighbors. Then $(x_1, x_2, ..., x_k)$ appear in that order on $C$, and the remaining nodes of $C$ were contracted in higher-level recursive calls by applications of Step 3 of Section 1.3.2.

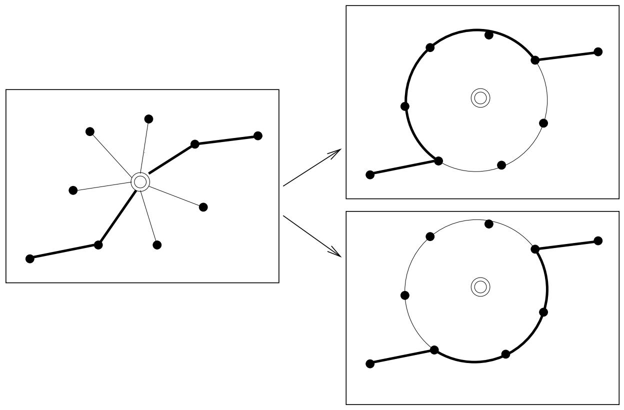


FIGURE 1.13: A C node $x$ of $G$ represents a cycle in $H$, so a path of $G$ through $x$ represents two possible paths in $H$.

Before proving this, let us examine what it implies about the relationship paths $G$ and paths in $H$. A path in $G$ through a $C$ node $x$ corresponds to two possible paths in $H$, one that proceeds in one direction about the cycle represented by $x$ and one that proceeds in the other direction (Figure 1.13).

**Proof** We give the construction of the lemma by induction on the depth of a call. The initial call is the base case, where there are no C nodes and the claim is vacuously true. For the induction step, let $x$ be the new C node created in the step, and let $C$ be the separating cycle bounding the full nodes of $G$ that is found in the step. Note that, as in Figure 1.12, $C$ may itself contain C nodes. Let $y$ be such a C node on $C$, and let $a, b$ be its neighbors on $C$. By induction, the lemma applies to $y$, so $y$ represents a cycle $C_y$ of $H$. The portion $(a, y, b)$ of $C$ represents two possible paths in $H$: one, $P_1$, that travels one way around $C_y$ avoiding empty neighbors of $y$ in $G$ (other than possibly $a$ or $b$), and another, $P_2$, that travels the other way, avoiding full neighbors of $y$ in $G$ (other than possibly $a$ or $b$). When constructing the cycle $C_x$ in $H$ represented by $x$, splice in $P_2$ in place of $(a, y, b)$. This ensures that the neighbors of $x$ in $G'$ will be on $C_x$ in $H$.

After application of the contraction step 3, some additional nodes of the cycle bounding the full subtrees are contracted out before they have a chance to become neighbors of the new C node, but these are those that the lemma allows to be removed. Therefore, the induction hypothesis to apply $x$ and $C_x$.

The constructive proof of the lemma show how the cycle represented by a C node can be found in $H$, by undoing the contractions while returning up through the recursive calls to $H$.

We now show how to return a subdivision of a $K_{3,3}$ or a $K_5$ when the conditions of Lemma 1.3 are not met.

Suppose first that the terminal edges of $T'_r$ do not form a path. Recall that a terminal edge is defined to be an edge whose removal from $T'_r$ separates it into two subtrees that each have both full and empty nodes. Clearly, the terminal edges are connected, so they form a subtree of $T'_r$ with at least three leaves, $z_1, z_2$, and $z_3$. Let $w$ be the meeting point of the paths of terminal edges that connect them, as illustrated in Figure 1.14. For $z_k \in \{z_1, z_2, z_3\}$, there is a path of non-terminal edges through full nodes, ending at a copy

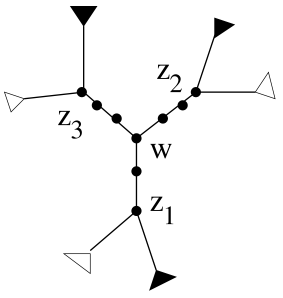


FIGURE 1.14: When the terminal edges of $T'_r$ do not form a path, they form a subtree of $T_C$ with at least three leaves, $z_1$, $z_2$, and $z_3$, each of which is adjacent to a full subtree that has a copy of $i$ and an empty subtree that has a copy of a proper ancestor of $i$. Let $w$ be the node of the tree from which paths to $z_1$, $z_2$, and $z_3$ branch.

of $i$, and a disjoint path of non-terminal edges through empty nodes, ending at a copy of a proper ancestor $t_k$ of $i$. Collectively, these paths correspond to paths in $G$ that are disjoint, except at their endpoints. (The multiple copies of $i$ are identified in $G$, and $\{t_1, t_2, t_3\}$ are not necessarily distinct.) There exists $t \in \{t_1, t_2, t_3\}$ of median height. The paths from $z_1$, $z_2$, and $z_3$ to $t_1$, $t_2$, and $t_3$ can be extended via edges of the DFS tree to paths to $t$ that are disjoint except at $t$ (Figure 1.15). These paths and the terminal edges joining them to $w$ define a subdivision of a $K_{3,3}$ of $G$ with bipartition $\{\{z_1, z_2, z_3\}, \{w, i, t\}\}$.

If $w$ is a P node, this $K_{3,3}$ of $G$ expands to to a subdivision of $K_{3,3}$ in $H$ by Lemma 1.5. If $w$ is a C node, but at least one of $z_1$, $z_2$, and $z_3$ fails to be a neighbor of $w$, then we can reduce this case to the previous one by taking into account that $w$ represents a cycle in $H$ by Lemma 1.5, and finding a P-node neighbor $w'$ of $w$ to serve in place of $w$, as illustrated in Figure 1.16. (Since no two C nodes are adjacent, $w'$ is a P node.)

Suppose $w$ is a C node and each of $z_1$, $z_2$, and $z_3$ is a neighbor of $w$. Without loss of generality, suppose $t_2$ is a minimal element of the not-necessarily distinct elements $t_1$, $t_2$, and $t_3$. If $t_2 = t_1$ or $t_2 = t_3$, then S&H returns a $K_5$; otherwise, the algorithm returns a $K_{3,3}$, as illustrated in Figure 1.17.

Finally, let us consider the case when there are at most two terminal nodes, but it is not possible to flip the full subtrees to one side of the terminal path and the empty trees to the other, due to constraints imposed some C node $x$ that lies on the terminal path.

For each neighbor $y$ of $x$, let $T_y$ be the neighboring subtree reachable from $x$ through $y$. That is, it is the component of the PC tree that contains $y$ when $x$ is deleted, and, conceptually, its "leaves" are the leaves of this subtree if it is then rooted at $y$. If $y$ lies on the terminal path, at least one of $T_y$'s leaves is a copy of $i$ and at least one is a copy of a proper ancestor of $y$. Otherwise, all of its leaves are copies of $i$ or all are copies of proper ancestors, depending on whether $T_y$ is full or empty.

The cyclic order of neighbors of $x$ blocks flipping the full and empty subtrees to opposite sides of the terminal path iff $x$ has four neighbors $a, b, c, d$ whose cyclic order about $x$ is $(a, b, c, d)$, and and where $T_a$ and $T_c$ each contain a full leaf and $T_b$ and $T_d$ each contain an empty leaf. (A neighbor on the terminal path can be selected for either of these two categories.)

Suppose that this is the case. In $T'_r$, there are disjoint paths from $a$ and $c$ through $T_a$ and $T_c$ to copies of $i$ and from $b$ and $d$ through $T_b$ and $T_d$ to copies of proper ancestors $t_b, t_d$ of $i$. These all correspond to paths of $G$ that are disjoint except at their endpoints. If $t_b = t_d$, let $t = t_b = t_d$. Otherwise, let $t$ be the lower of the two. The path to the other of the two

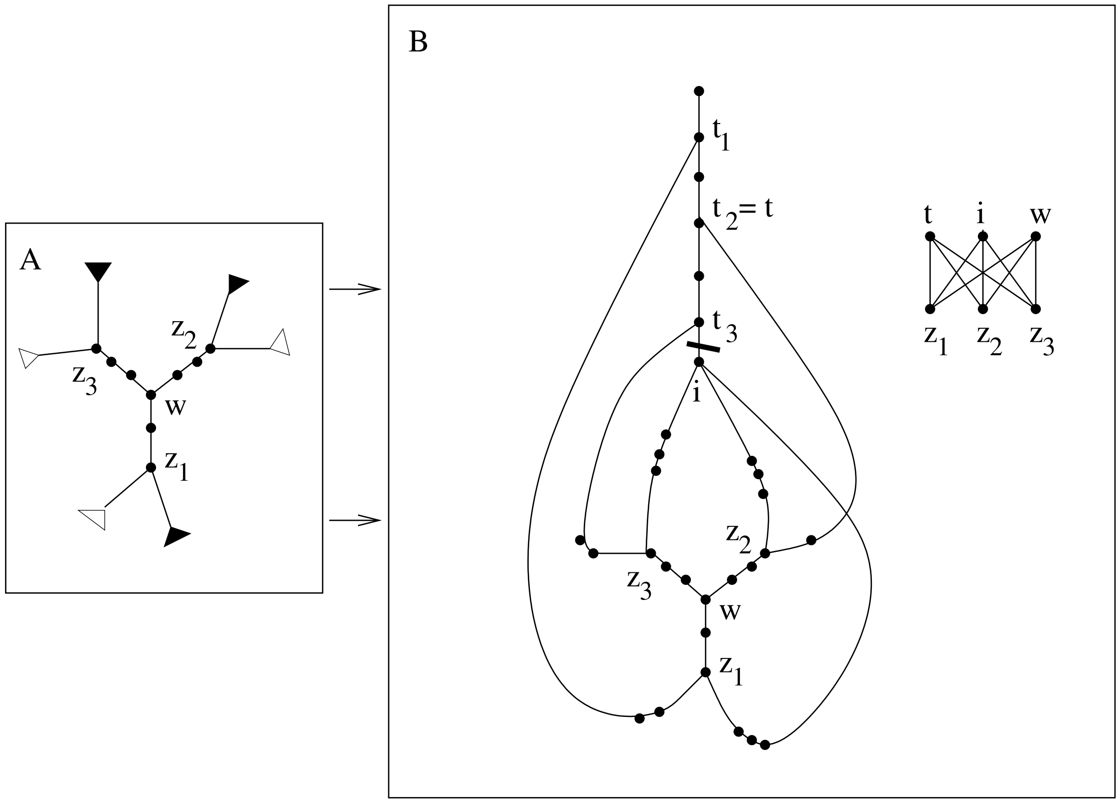


FIGURE 1.15: If $w$ is a P node, S&H gets a $K_{3,3}$ with bipartition $\{\{z_1, z_2, z_3\}, \{w, i, t\}\}$. This $K_{3,3}$ is the contraction of a $K_{3,3}$ of the original input graph $H$.

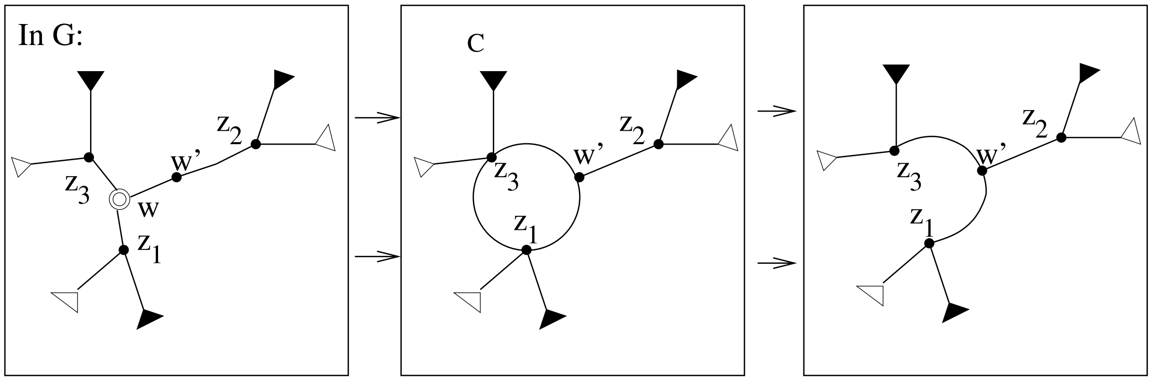


FIGURE 1.16: If $w$ is a C node but at least one of $z_1$, $z_2$, and $z_3$ is a non-neighbor of $w$, the case can be reduced to that of Figure 1.15 by selecting a neighbor $w'$ to serve in the role of $w$.

can be extended by DFS tree edges to $t$ that is still disjoint from the other paths, except at $t$. These paths, the cycle represented by $x$, and the DFS tree edges between $t$ and $i$ give

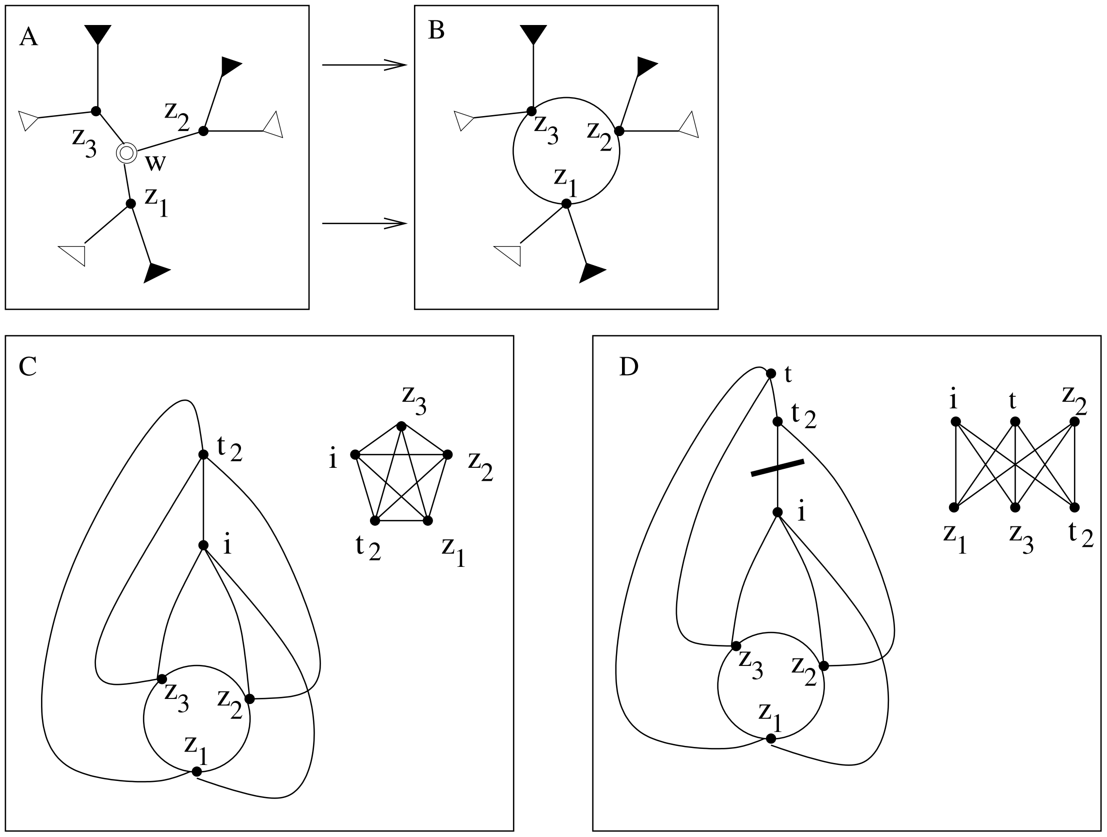


FIGURE 1.17: Each of $z_1, z_2$ and $z_3$ is a neighbor of $w$, which is a C node. Without loss of generality, suppose $t_2$ is a minimal element of $t_1$, $t_2$, and $t_3$. If one of the others, say, $t_3$, is equal to $t_2$, then the algorithm returns a $K_5$. Otherwise, it returns a $K_{3,3}$.

rise to a $K_{3,3}$ of $H$ by Lemma 1.5, as in Figure 1.18.

In addition to giving an algorithm for returning a subdivision of a $K_{3,3}$ or a $K_5$ when the embedding algorithm fails, we have just proven Lemma 1.3, since no planar graph contains such a subdivision.

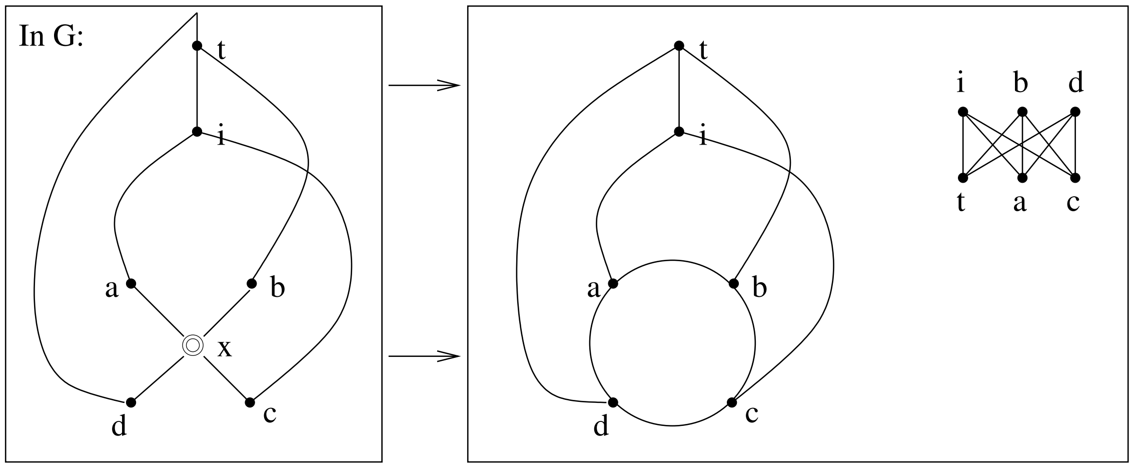


FIGURE 1.18: If a C node $x$ has four neighbors $(a, b, c, d)$ in that order, such that there are disjoint paths from $a$ and $c$ to $i$ and $b$ and $d$ to an ancestor of $i$, then the cycle that $x$ represents, together with these paths, gives a $K_{3,3}$.

## 1.4 Acknowledgment

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

## References

[1] K. Appel and W. Haken. Every planar map is four colorable. part i. discharging. *Illinois J. Math.*, 21:429–490, 1977.

[2] K. Appel and W. Haken. Every planar map is four colorable. part ii. reducibility. *Illinois J. Math.*, 21:491–567, 1977.

[3] S. Benzer. On the topology of the genetic fine structure. *Proc. Nat. Acad. Sci. U.S.A.*, 45:1607–1620, 1959.

[4] K. S. Booth. *PQ-Tree Algorithms.* PhD thesis, Department of Computer Science, U.C. Berkeley, 1975.

[5] S. Booth and S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. Syst. Sci.*, 13:335–379, 1976.

[6] J. Boyer and W. Myrvold. 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*, 10:140–149, 1999.

[7] J. Cai, X. Han, and R. E. Tarjan. An $o(mlogn)$-time algorithm for the maximal planar subgraph problem. *SIAM J. Comput.*, 22:1142–1162, 1993.

[8] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using pq-trees. *J. Comput. Syst. Sci.*, 30:54–76, 1985.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* McGraw Hill, Boston, 2001.

[10] D. R. Fulkerson and O. Gross. Incidence matrices and interval graphs. *Pacific J. Math.*, 15:835–855, 1965.

[11] M. Habib, R. M. McConnell, C. Paul, and L. Viennot. Lex-bfs and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theoretical Computer Science*, 234:59–84, 2000.

[12] F. Harary. *Graph Theory.* Addison-Wesley, Reading, Massachusetts, 1969.

[13] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *J. Assoc. Comput. Mach.*, 21:549–568, 1974.

[14] W. L. Hsu. PC-trees vs. PQ-trees. *Lecture Notes in Computer Science*, 2108:207–217, 2001.

[15] W. L. Hsu. A simple test for the consecutive ones property. *Journal of Algorithms*, 42:1–16, 2002.

[16] W. L. Hsu and T. Ma. Fast and simple algorithms for recognizing chordal comparability graphs and interval graphs. *SIAM J. Comput.*, 28:1004–1020, 1999.

[17] W. L. Hsu and R. M. McConnell. PC trees and circular-ones arrangements. *Theoretical Computer Science*, 296:59–74, 2003.

[18] P. N. Klein and J. H. Reif. An efficient parallel algorithm for planarity. *Journal of Computer and System Science*, 18:190–246, 1988.

[19] N. Korte and R. H. Moehring. An incremental linear-time algorithm for recognizing interval graphs. *SIAM J. Comput.*, pages 68–81, 1989.

[20] D. Kratsch, R. M. McConnell, K. Mehlhorn, and J. P. Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 866–875, 2003.

[21] C. Kuratowski. Sur le problème des courbes gauches en topologie. *Fundamata Mathematicae*, pages 271–283, 1930.

[22] C. Lekkerker and D. Boland. Representation of finite graphs by a set of intervals on the real line. *Fund. Math.*, 51:45–64, 1962.

[23] W. F. Lu and W.-L. Hsu. A test for interval graphs on noisy data. *Lecture Notes in Computer Science*, 2467:196-208, 2003.

[24] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In P. Rosenstiehl, editor, *Theory of Graphs*, pages 215–232. Gordon and Breach, New York, 1967.

[25] R. M. McConnell. A certifying algorithm for the consecutive-ones property. *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA04)*, 15:761–770, 2004.

[26] R. M. McConnell and J. P. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, 5:536–545, 1994.

[27] R. M. McConnell and J. P. Spinrad. Linear-time transitive orientation. *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, 8:19–25, 1997.

[28] R. M. McConnell and J. P. Spinrad. Modular decomposition and transitive orientation. *Discrete Mathematics*, 201(1-3):189–241, 1999.

[29] R. M. McConnell. Linear-time recognition of circular-arc graphs. *Algorithmica*, 37:93–147, 2003.

[30] K. Mehlhorn. Graph algorithms and NP-completeness. *Data structures and algorithms*, 2:93–122, 1984.

[31] K. Mehlhorn and P. Mutzel. On the embedding phase of the hopcroft and tarjan planarity testing algorithm. *Algorithmica*, 16:233–242, 1996.

[32] D. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5:266–283, 1976.

[33] 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*, pages 110–122, 1993.

[34] W. K. Shih and W. L. Hsu. A new planarity test. *Theoretical Computer Science*, 223:179–191, 1999.

[35] J. Small. A unified approach to testing, embedding and drawing planar graphs. *Proc. ALCOM International Workshop on Graph Drawing, Sevre, France*, 1993.

[36] H. Stamm-Wilbrandt. A simple linear-time algorithm for embedding maximal planar graphs. *Proc. ALCOM International Workshop on Graph Drawing, Sevre, France.*, 1993.

[37] R. Thomas. Planarity in linear time – lecture notes, georgia institute of technology. *Available at www.math.gatech.edu/∼thomas/*, 1997.

[38] A. Tucker. An efficient test for circular-arc graphs. *SIAM Journal on Computing*, 9:1–24, 1980.

[39] M. S. Waterman. *Introduction to Computational Biology: Maps, Sequences and Genomes.* Chapman and Hall, New York, 1995.