

3

Planarity Testing and Embedding

	3.1 Properties and Characterizations of Planar Graphs.....	2
	Basic Definitions • Properties • Characterizations	
	3.2 Planarity Problems	3
	3.3 History of Planarity Algorithms.....	3
	3.4 Common Algorithmic Techniques and Tools	4
	3.5 The Path-Addition Approach	4
	Initialization Phase • Path Decomposition Phase • Path Embedding Phase	
Pier Francesco Cortese <i>Universita' di Roma Tre</i>	3.6 The Vertex-Addition Approach	5
	Lempel, Even, and Cederbaum • PQ-Trees	
Maurizio Patrignani <i>Universita' di Roma Tre</i>	3.7 The Block Embedding Approach	9
	Shih and Hsu algorithm • Boyer and Myrvold algorithm	

Testing the planarity of a graph and possibly drawing it without intersections is one of the most fascinating and intriguing problems of the graph drawing and graph theory areas. Although the problem *per se* can be easily stated, and a complete characterization of planar graphs was available since 1930, an efficient solution to it was found only in the seventies of the last century.

Planar graphs play an important role both in the graph theory and in the graph drawing areas. In fact, planar graphs have several interesting properties: for example they are sparse, four-colorable, allow a number of operations to be performed efficiently, and their structure can be elegantly described by an SPQR-tree (see Section 3.1.2). From the information visualization perspective, instead, as edge crossings turn out to be the main culprit for reducing readability, planar drawings of graphs are considered clear and comprehensible. As a matter of fact, the study of planarity has motivated much of the development of graph theory.

In this chapter we review the number of alternative algorithms available in the literature for efficiently testing planarity and computing planar embeddings. Some of these algorithms already attracted the attention of book writers (as for example the algorithm by Hopcroft and Tarjan [HT74] which is fully described by Even [Eve79]). Some other are only recently acquired by the research community. To these we dedicate more attention.

The chapter is organized as follows: Section 3.1 introduces basic definitions, properties, and characterizations for planar graphs; Section 3.2 formally defines the planarity testing and embedding problems; Section 3.3 follows an historic perspective to introduce the main algorithms and a conventional classification for them. Some algorithmic techniques are common to more than one algorithm and not rarely to all of them. These are collected in Section 3.4. Finally, the three Sections 3.5, 3.6, and 3.7 are devoted to the three main

approaches to the planarity problem, namely the “path addition,” “vertex addition,” and “block embedding” approaches, respectively.

3.1 Properties and Characterizations of Planar Graphs

3.1.1 Basic Definitions

A (simple) graph $G(V, E)$ is an ordered pair consisting of a finite set of *vertices* V and finite set E of *edges*, that is, pairs (u, v) of distinct vertices. If edge $(u, v) \in E$, vertices u and v are said *adjacent* and (u, v) is said to be *incident* to u and v . Two edges are *adjacent* if they have a vertex in common. If each edge is an unordered (ordered) pair of vertices, then the graph is *undirected* (*directed*).

A *drawing* Γ of a graph G maps each vertex v to a distinct point $\Gamma(v)$ of the plane and each edge (u, v) to a simple open Jordan curve $\Gamma(u, v)$ with endpoints $\Gamma(u)$ and $\Gamma(v)$. A drawing is *planar* if no two distinct edges intersect. A graph is *planar* if it admits a planar drawing. A planar drawing partitions the plane into connected regions called *faces*. The unbounded face is usually called *external face*. Given a planar drawing, the (clockwise) circular order of the edges incident to each vertex is fixed. Two planar drawings are *equivalent* if they determine the same circular orderings of the edges incident to each vertex. A (*planar*) *embedding* is an equivalence class of planar drawings and is described by the clockwise circular order of the edges incident on each vertex.

An undirected graph G is *connected* if, for each pair of nodes u and v , G contains a path from u to v . If a graph is not connected, we call *connected component* its maximal connected subgraphs.

A vertex $v \in V$ is a *cut vertex* if its removal makes the graph disconnected. A graph is *biconnected* if it does not contain cut vertices. A maximal biconnected subgraph in a non-biconnected graph is a *biconnected component*, or *block*. Note that a cut vertex belongs to several blocks.

A non-connected graph is planar if and only if all its connected components are planar. Thus, in the following, without loss of generality, we only consider connected graphs. Also, a planar embedding of a graph implies a planar embedding for each one of its blocks and, on the contrary, starting from a planar embedding of the blocks, a planar embedding for the whole graph can be found. Thus, since the blocks can be identified in linear time [Tar72], a common strategy, both to test planarity and to compute a planar embedding, is that of dividing the graph into its blocks, and tackle each block separately.

A (simple) *cycle* is a circular list of distinct, mutually adjacent, vertices.

A (directed, rooted) *tree* T is a directed graph with one distinguished vertex, called the *root* r such that every vertex in T is reachable from r , no edges enter r , and exactly one edge enters every other vertex in T .

3.1.2 Properties

Planar graphs have a variety of properties whose exploitation allows to efficiently perform a number of operations on planar graphs and to obtain a succinct representation of their inner structure.

Perhaps the most important property from an efficiency perspective is given by the Euler Theorem, which states that planar graphs are sparse. Namely, given a plane graph with n vertices, m edges and f faces, we have $n - m + f = 2$. A simple corollary is that for a maximal planar graph, where each face is a triangle, we have $m = 3n - 6$, and therefore, for any graph with at least three vertices, we have $m \leq 3n - 6$.

A graph $G(V, E)$ is k colorable if its vertices can be partitioned into k sets V_1, V_2, \dots, V_k in such a way that no edge is incident to two vertices of the same set. The Four Color Theorem [AH77, AHK77, RSST97] asserts that any planar graph is four colorable and settles a conjecture that was for a more than a century the most famous unsolved problem in graph theory and perhaps in all of mathematics [Har69]. Now that the Four Color Conjecture is proved to be true, apart from being considered an important property of planar graph, it has been mentioned as the most notable property of the number four.

3.1.3 Characterizations

The first complete characterization of planar graphs is due to Kuratowsky [Kur30], and states that A graph is planar if and only if it contains no subgraph that is a subdivision of K_5 or $K_{3,3}$, where K_5 is the complete graph of order 5 and $K_{3,3}$ is the complete bipartite graph with 3 vertices in each of the sets of the partition. An equivalent later results, recasted in terms of graph minors is Wagner's theorem that states that a graph is planar if and only if it does not contain K_5 or $K_{3,3}$ as a minor [Wag37a, HT65]. If the graph is 3-connected a simpler characterization can be formulated. Namely, a 3-connected graph distinct from K_5 is planar if and only if it contains no subgraph that is a subdivision of $K_{3,3}$ [Wag37b].

Alternative characterizations can be found in the literature based on DFS traversals of the graph [dR85], or on the existence of a dual graph.

3.2 Planarity Problems

- Testing planarity
- Embedding a planar graph
- Planarization of non-planar graphs [Lie01]
- Other related problems: on-line (dynamic) planarity testing, max plan, kuratowsky subgraph isolation, constrained planarity, etc.

3.3 History of Planarity Algorithms

In 1974 Hopcroft and Tarjan [HT74] proposed the first linear-time planarity testing algorithm. This algorithm, also called “path-addition algorithm,” starts from a cycle and adds to it one path at a time. However, the algorithm is so complex and difficult to implement that several other contributions followed their breakthrough. For example, about twenty years after [HT74], Mehlhorn and Mutzel [MM96] contributed a paper to clarify how to construct the embedding of a graph that is found to be planar by the original Hopcroft and Tarjan algorithm.

A different approach has its starting point in the algorithm presented by Lempel, Even, and Cederbaum [LEC67]. This algorithm, also called “vertex addition algorithm,” is based on considering the vertices one-by-one, following an st -numbering; it has been shown to be implementable in linear time by Booth and Lueker [BL76]. Also in this case, a further contribution by Chiba, Nishizeki, Abe, and Ozawa [CNAO85] has been needed for showing how to construct an embedding of a graph that is found planar.

A further interesting algorithm is based on a characterization given by de Fraysseix and Rosenstiehl [dR85]. This algorithm has not been fully described in the literature but has a very efficient implementation in the Pigale software library [dO].

However, although the planarity problem has been carefully studied in the above cited literature, the story of the planarity testing algorithms enumerates several more recent contributions. The motivations behind such relatively new papers are two-fold. On one side, even if the known algorithms are combinatorially elegant, they are quite difficult to understand and to implement. On the other side, the researchers are interested in deepening the relationships between planarity and Depth First Search (DFS). Such relationships are clearly strong but, probably, up to now, not completely understood.

Two recent DFS-based planarity testing algorithms, whose similarities were stressed in [Tho99], are those presented by Shih and Hsu [SH93, SH99, Hsu03] and by Boyer and Myrvold [BM99, BM04].

Shih and Hsu algorithm replaces biconnected portions of the graph with single nodes, called C -nodes, whose embedding is fixed. However, up to now, it is unclear that their algorithm can be implemented in linear time as it is described in the paper.

Boyer and Myrvold algorithm represents embedded biconnected portions of the graph with a data structure that allows the embeddings to be “flipped” in constant time.

More recently, Boyer and Myrvold [BM04] presented an algorithm that, although inspired by [BM99], constitutes, in our opinion, a new and original approach to the planarity testing problem.

3.4 Common Algorithmic Techniques and Tools

In this section we introduce some definitions and common techniques used by the planarity testing algorithms. The most important technique, common to almost all the algorithms, is the *Depth First Search*, or DFS in short. The DFS is a method for visiting all the vertices of a graph G . It starts from a arbitrarily chosen vertex of G , and continues moving from the current vertex to an adjacent one, until unexplored vertices can be found. When the current vertex has no unexplored adjacent vertices, the traversal backtracks to the first vertex with unexplored adjacent vertices.

The edges used by the DFS to reach the vertices of G form a spanning tree T of G , called *Palm Tree*, or *DFS Tree*. The root of T is the vertex from which the visit is started. The edges of T are called *tree edges*, while the remaining edges of G are called *back edges*.

After performing a DFS traversal, each vertex v of G can be associated with a *DFS index*, $DFS(v)$, that is the order in which v was reached during the DFS visit. The root of T has index one. For a tree edge (u, v) , we have that $DFS(u) < DFS(v)$. On the contrary, a back edge is supposed to be oriented from the end vertex with higher DFS index to the end vertex with lower DFS index. An example of DFS is shown in Fig. 3.4.

For each vertex v of G can be also defined two sets of edges, called $B_{in}(v)$ and $B_{out}(v)$. These sets contain respectively the back edges entering and exiting v . Note that each back edge in $B_{in}(v)$ connects v to a descendant in the DFS tree, while each back edge in $B_{out}(v)$ connects v to an ancestor. At last, the *lowpoint* of a vertex v , denoted by $Lowpt(v)$, is the lower DFS index of an ancestor of v reachable through a back edge from a descendant of v .

3.5 The Path-Addition Approach

In the seminal paper by Hopcroft and Tarjan [HT74] a linear time planarity testing algorithm was presented. Although the algorithm inner working is based on the search for a planar representation of the input graph, how to actually produce a planar embedding of it was not described. Twenty years later, Mehlhorn and Mutzel filled the gap, describing

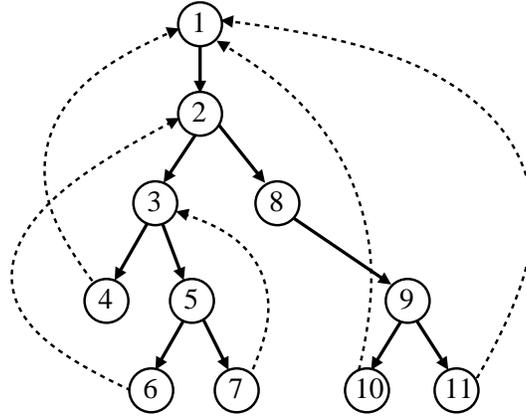


Figure 3.1 A DFS traversal of a graph. Thick lines represent the tree edges, while the back edges are drawn with dashed lines. Each vertex is identified with its DFS index.

how to collect, during the testing phase, the information needed to construct the planar embedding that was implicitly used to show planarity [MM96].

The graph is supposed to be biconnected.

3.5.1 Initialization Phase

During this phase a DFS-tree is computed for the biconnected input graph. Also $L(v)$ and $L2(v)$ are computed for each vertex.

3.5.2 Path Decomposition Phase

The purpose of this phase is to decompose the input graph into a collection of paths. These paths are sorted in such a way that:

- the first path P_1 is actually a (simple) cycle
- each path P_i , $i > 1$ is simple and it has only the first and last vertex in common with previously generated paths.

3.5.3 Path Embedding Phase

The paths obtained from the previous phase are considered one at a time in the order in which they are sorted.

3.6 The Vertex-Addition Approach

3.6.1 Lempel, Even, and Cederbaum

The Lempel, Even, and Cederbaum algorithm tests the planarity of a biconnected graph.

St-Numbering

This algorithm is based on the *st-numbering*, that is, a technique, that assigns an index to each vertex. More precisely, given a biconnected graph $G(V, E)$ an *st-numbering*

on G is a function $f : V \rightarrow N$, the *st-function*, that assigns to each vertex a different index, according to the following rules:

1. two adjacent vertices are arbitrarily chosen, called s and t ;
2. $f(s) = 1$, $f(t) = n$, where n is the number of nodes;
3. for each $x \in V$, $x \neq s, t$ there are at least two vertices y and z , adjacent to x , such that $f(y) > f(x) > f(z)$.

It is always possible to find an *st-numbering* for a biconnected graph; see [ET76] for further details. An *st-graph* is a biconnected graph and the associated st-numbering. An st-graph is a directed graph, where each edge is oriented from the vertex with the lower value to the vertex with higher value. In the following of this section, the vertices of an st-graph will be identified with their indexes in the st-numbering.

Bush Form

Given a st-graph G , let G_k the subgraph induced by the vertices $1 \dots k$. The following Lemma shows an important property of st-graphs, exploited by the planarity testing algorithm:

LEMMA 3.1 If G is a planar st-graph, there exists a planar embedding of G_k such that the edges in $G - G_k$ are drawn in the external face.

Starting from G_k , it is possible to define the graph B_k : G_k is augmented with the edges adjacent to the node of the external face. These edges are called *virtual edges*, while the leaves that they introduce in B_k are *virtual vertices*. Virtual vertices are labeled with the same indexes in G , but they are kept separate in B_k . According to Lemma 3.1, it is possible to find a planar embedding for B_k such that all the virtual edges and vertices are on the external face. Such an embedding is called *bush form*. An example of bush form is shown in Fig. 3.6.1.

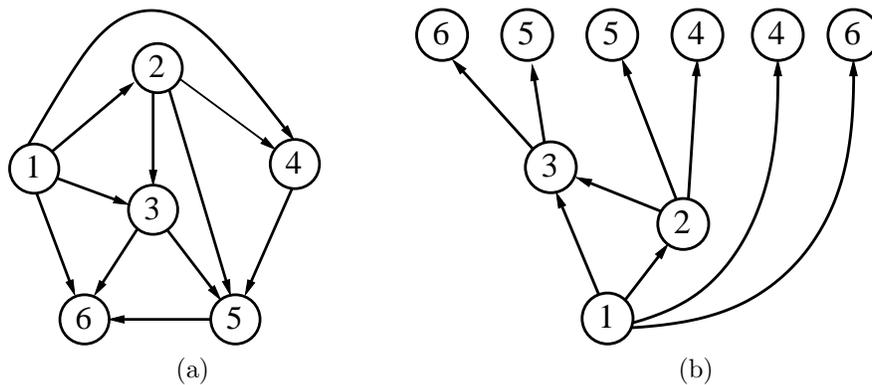


Figure 3.2 Example of bush form;(a) st-graph, (b) a bush form of the graph B_3 .

A bush form is usually represented drawing all the virtual vertices on an horizontal line. Given a bush form, we call *reduction* the operation that dispose consecutively the virtual vertices with the same label. This operation is performed through permutations and flips of the virtual vertices. Consider a cut vertex v of a bush form of B_k , with $K > 1$; each

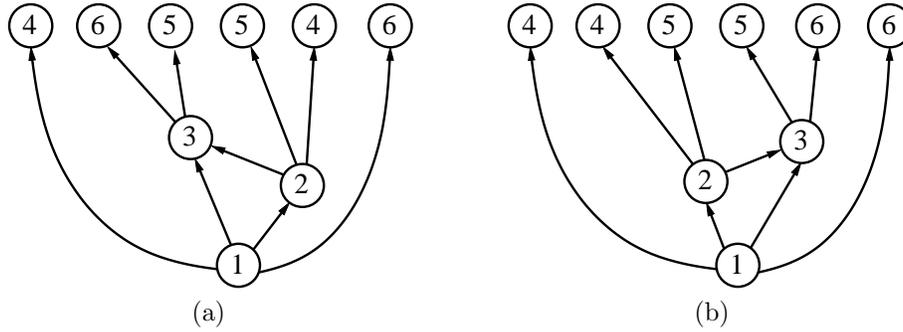


Figure 3.3 Operations on the bush form of fig. 3.6.1.b: permutation (a) and flipping(b).

component of B_k containing v , except the one that contains vertex 1, is in its turn a bush form. These components can be permuted around v , obtaining different bush forms for B_k . If $k = 1$, all the components can be permuted. Moreover, each of these components can be flipped, positioning the virtual vertices in reverse order. Figure 3.6.1 illustrates an example of permutations and flipping in a bush form.

The following theorem summarizes the main property of the bush forms:

Theorem 3.1 *If B_k^1 and B_k^2 are two different bush form of B_k , then there exist a sequence of permutations and flips that transform B_k^1 in the bush form B_k^3 , such that the virtual vertices in B_k^2 and B_k^3 appear in the same order.*

From this theorem follows immediately the following results:

COROLLARY 3.1 *If G is planar and B_k^1 is a bush form of B_k , then there exist a sequence of permutations and flips which transforms B_k^1 in B_k^2 , in which all the virtual vertices with the label $k + 1$ appear consecutively.*

The Algorithm

The planarity testing algorithm works as follows. First, an st-numbering is computed on the input graph G , and then the vertices are proceeded according to their index. For each vertex v a corresponding bush form B_v is realized: B_1 is composed by vertex 1 and all its adjacent vertices; each bush form B_k , $k \geq 2$ is realized starting from B_{k-1} , by performing the reduction and expansion operations. The purpose of the reduction operations on B_v is to obtain a new bush form such that all the virtual vertices in B_v with label $v + 1$ appear consecutively. If the graph is planar, according to Corollary 3.1, this operation is always possible. The sequence of permutations and flips which must be applied to perform a reduction and the representation of the bush form is fully described in [LEC67]. However, a more effective representation of the virtual vertices and their related operations can be obtained with the *PQ-Trees* data structure, described below in this section.

After performing a reduction operation on a bush form of B_k , the algorithm replace all the virtual vertices with label $k + 1$ with a single vertex. Finally, a bush form for B_{k+1} can be obtained augmenting the new graph with the virtual vertices adjacent to $k + 1$ which were not present in B_k . If it is not possible to perform a reduction on a bush form, then the graph G is not planar. The algorithm ends and states that G is planar if it is possible to construct a bush form for B_n . Note that B_n coincides with the input graph G . This property is stated in Theorem 3.2.

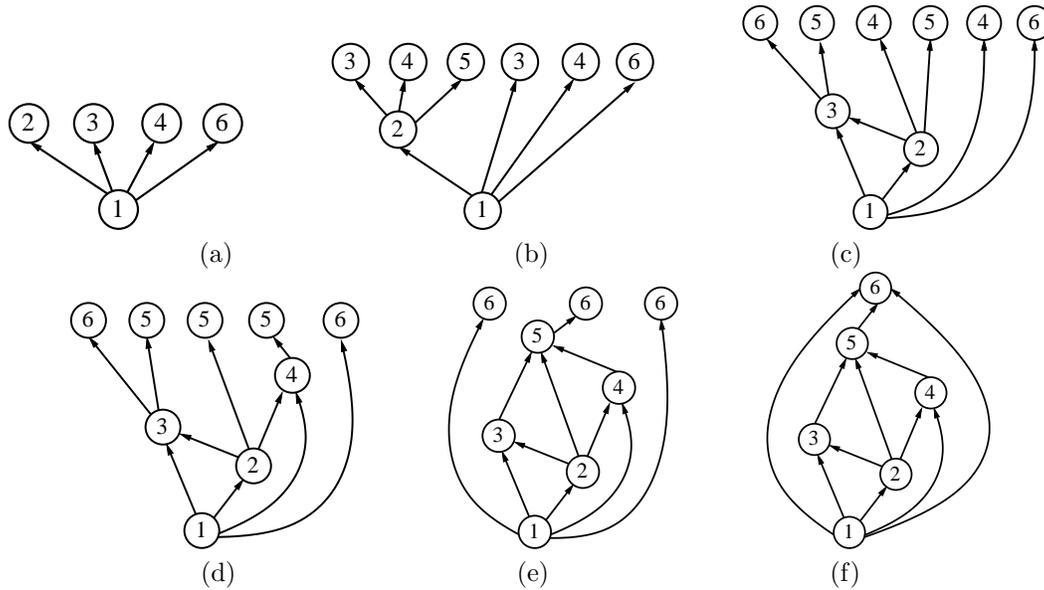


Figure 3.4 The planarity testing algorithm is applied to the st-graph in figure 3.6.1.a

Theorem 3.2 *An st-graph G is planar if and only if for each vertex k , $k \in \{1 \dots n\}$, it admits a bush form B_k .*

An example of application of the planarity testing algorithm on the st-graph of figure 3.6.1.a is shown in figure 3.6.1.

3.6.2 PQ-Trees

The PQ-tree data structure was introduced by Booth and Lueker [BL76], and it is exploited in the planarity testing algorithm to obtain a linear time implementation. A PQ-tree represents in an implicit way all the admissible permutations of a finite set of elements S , respecting some constraints on the elements. The constraints that can be imposed specify that the elements of a subset of S must appear consecutively in any admissible permutation.

For example, if $S = \{1, 2, 3, 4, 5, 6\}$, and there are two constraints $S_1 = \{1, 2, 3\}$ and $S_2 = \{3, 6\}$, then $P_1 = (2, 1, 3, 6, 5, 4)$ is an admissible permutation for S , while $P_2 = (1, 2, 3, 5, 6, 4)$ is not admissible.

The leaves of a PQ-tree correspond to the elements of S , while there are two different types of internal nodes. A node P indicates that its children can be permuted in any order, while a node Q forces the children in a specific order, and they can only be flipped. An example of PQ-tree is depicted in Fig. 3.6.2.

The operations defined on the PQ-trees permit the update of the structure, with the specification of a set of constraints, in amortized linear time. PQ-trees are utilized in planarity testing because they can easily represent the set of virtual vertices of the bush forms, and the constraints defined on their permutations. An efficient implementation of the algorithm can be obtained by representing and updating at each step only the set of the virtual vertices, and not the entire bush forms. Thus, this technique leads to a linear time implementation of the Lempel, Even and Cederbaum algorithm.

In 1985 Chiba, Nichizeki, Abe, and Ozawa [CNAO85] proposed some modifications of the PQ-tree structure and of the testing algorithm, producing a new linear time algorithm that computes a planar embedding of a planar graph.

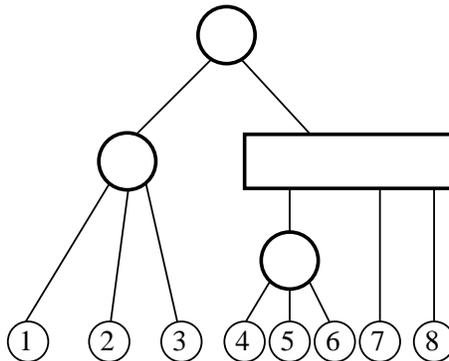


Figure 3.5 An example of PQ-tree defined on the set $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$. Circles represent P nodes, while the rectangle represents a Q node.

3.7 The Block Embedding Approach

In this section we describe two more recent algorithms due to Shih and Hsu [SH99] and Boyer and Myrvold [BM99]. The two algorithms are so similar that they have been sometimes identified [Tho99]. A slightly different version of the Boyer and Myrvold's algorithm has been presented by the same authors in [BM04].

These algorithms test the planarity of a graph in linear time by trying to construct a planar embedding of it. The basic idea is to construct a DFS-tree of the graph, and then to add the back edges, constructing incrementally a planar embedding (if it exists). During the construction of the planar embedding, a *partial embedding* of the graph is maintained, as explained in the following. The partial embedding refers only to a subset of the edges of the graph. At the very beginning of the test, when no back edges are still added to the embedding, the embedded graph is composed only by a set of elementary blocks, each constituted by a single tree edge. The addition of the back edges can merge the blocks between them, producing larger biconnected components. At any iteration of the algorithms we have that the embedding of each block is fixed, while the blocks can be flipped and permuted around their cut vertices. Hence, the embedding of the whole graph is partially specified for the cut-vertices, while it is fixed for the other vertices. For these reasons, the approach used in all these algorithms can be defined as *block embedding*. However, it could be also possible to classify the same algorithms as vertex addition or edge addition approaches; a different definition can be argued if we considered the operations performed by the algorithms from different points of view. Both the algorithms consider the vertices one after the other, in reverse DFI order, adding to the partial embedding the set of back edges incident to the current vertex; hence they may be defined as vertex addition algorithms. However, the partial embedding is modified by the addition of the back edges, and then it is also possible to define the algorithms as edge addition. We choose to define these algorithms as block construction approaches, because we want to underline the main difference of these algorithms with the previous approaches, that is the construction of the blocks and the partial embedding of the graph.

The block embedding approach consists of three phases: *Preprocessing*, *Embedding Construction*, and *Embedding Reporting*. The preprocessing phase performs a DFS traversal of the graph and initializes the data structures with the edges of the spanning tree. During the preprocessing phase $DFS(v)$, $L(v)$ and the set $B_{in}(v)$ is computed for each node v . The embedding construction phase performs a *step* for each node v , taken in inverse DFS

order. This corresponds to a post-order traversal of the DFS-Tree. Let v be the current visited vertex and let $G(v)$ be the subgraph of G composed by the edges of the DFS tree augmented with the edges in $B_{in}(w)$, for each w such that $DFS(w) \geq v$. The target of step v is to determine (if it is possible) a planar embedding for the blocks of $G(v)$ exploiting the planar embeddings already computed for graph $G(u)$, where $DFS(u) = DFS(v) + 1$, which are preserved up to a “flip” operation. See Figs. 3.7 and 3.7. In order to make this possible, the embeddings of the blocks of $G(v)$, computed when vertex v is processed, must have the outer vertices of $G(v)$ on the external face. We call *outer vertices* of $G(v)$ the cutvertices of $G(v)$ and the vertices of $G(v)$ incident to a back-edge that is not in $G(v)$.

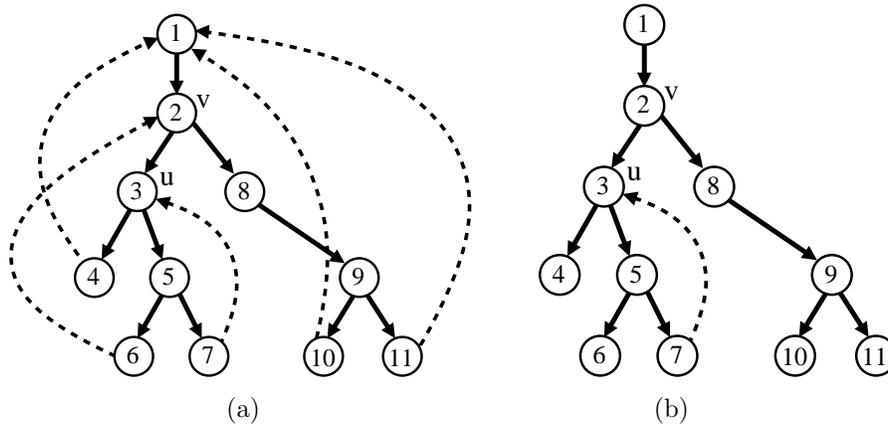


Figure 3.6 (a) A graph G to be planarized; the edges of the DFS tree are drawn thick. $DFS(u) = DFS(v) + 1$. (b) Graph $G(u)$.

Observe that: (i) $G(u)$ is a subgraph $G(v)$. (ii) The edges belonging to $G(v)$ that do not belong to $G(u)$ are exactly the back-edges in $B_{in}(v)$. (iii) Since the input graph G is biconnected, when $DFS(v) = 1$, $G(v) = G$ has a single block, and a planar embedding of $G(v)$, if one exists, is a planar embedding for G .

3.7.1 Shih and Hsu algorithm

Shih and Hsu algorithm.

PC-Trees

Description of the PC-Tree data structure

3.7.2 Boyer and Myrvold algorithm

The algorithm is defined according to the three phases of the block construction approach: *Preprocessing*, *Embedding Construction*, and *Embedding Reporting*.

In the preprocessing phase, an elementary block is associated with each edge of the DFS-Tree, and $DFS(v)$, $L(v)$, and $B_{in}(v)$ are computed for each vertex. The elementary blocks are represented with the data structure described in the following of this section.

Each step of the embedding construction phase, performed with respect to a specific vertex v , is composed by two strictly related tasks, called *Walk-up* and *Walk Down*. The

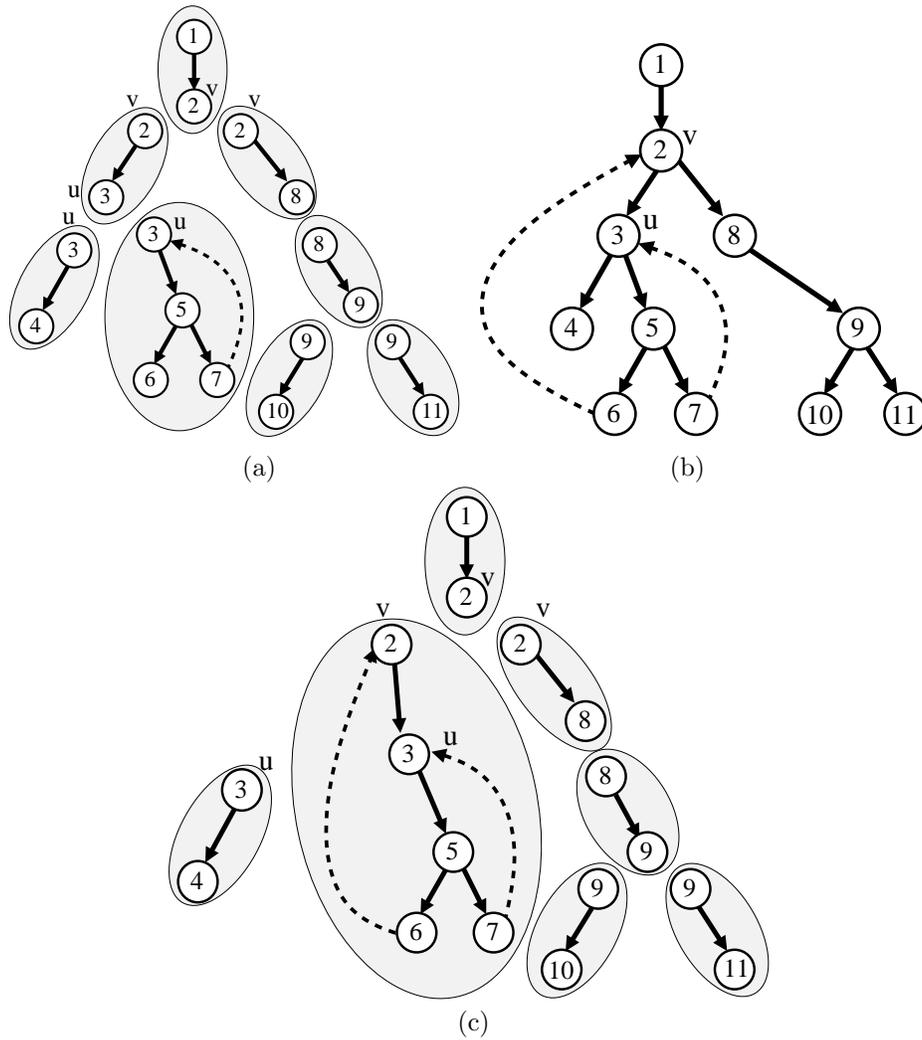


Figure 3.7 (a) An embedding of the blocks of graph $G(u)$ of Fig. 3.7. (b) Graph $G(v)$. (c) An embedding of the blocks of $G(v)$.

purpose of Walk-up is to test if the back-edges in B_{in} can be inserted without violating planarity. Walk-up searches a path, along the external border of the blocks, for each back edge in $B_{in}(v)$. Walk-Down inserts the back edges, updating the data structure and eventually merging the blocks. Walk Down on node v is performed only if a path for each back edge in $B_{in}(v)$ was found by the previous walk-up.

The remainder of this section describes the walk-up and the walk-down tasks.

The Walk-Up Task

The purpose of the walk-up task is to verify if there is a way to add the back-edges in $B_{in}(v)$ to the embedded blocks of $G(u)$ in such a way that the embedded blocks of $G(v)$ have the outer vertices of $G(v)$ on the external face. Also, during the walk-up task suitable information is collected in order to implicitly describe the embeddings which are actually built in the successive walk-down task.

The walk-up task works as follows. For each edge $e = (w, v)$ in $B_{in}(v)$ a path p_e is searched (and marked) from w to v along the blocks that contain an edge in $S(e)$. The paths must satisfy several constraints.

In order to describe these constraints, we need to introduce some definitions. Consider an embedded block b of $G(u)$ containing some edges of $S(e)$. Observe that the edges of $S(e)$ contained in b form a subpath of $S(e)$. This subpath has its endpoints on the external face of b . We call these two vertices *root of b* and *entry point of b* with respect to $S(e)$, the former (latter) being the one nearest to (farthest from) the root of the DFS tree. Observe that each $S(e)$ traversing b may have a different entry point $v_{b,e}$, but all have the same root vertex r_b , which is the vertex of the block b with the lowest DFS index.

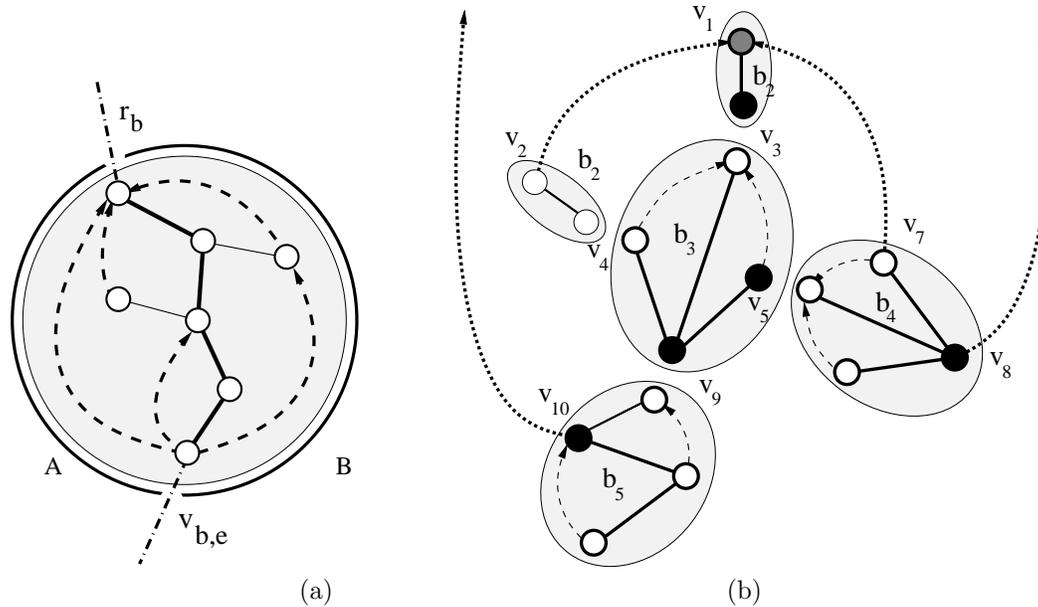


Figure 3.8 (a) A block b whose intersection with the support $S(e)$ (drawn with solid lines) is not empty. The two borders \mathcal{A} and \mathcal{B} of b from the entry point $v_{b,e}$ to the root r_b of the support $S(e)$ are drawn with solid lines along the boundary of b . (b) If vertex v_1 is the current processed vertex, and if the edges shown with dotted lines are the only back-edges in $B_{in}(v_1)$, then vertex v_{10} is externally active for block b_5 ; vertices v_5 and v_9 are externally active for block b_3 , vertex v_8 is externally active for block b_4 ; and vertex v_3 is externally active for block b_1 . Vertices v_1 , v_8 , v_9 , and v_{10} are also outer vertices. Externally active vertices are drawn black. Vertices v_1 , v_8 , v_9 , and v_{10} are also outer vertices.

Given a specific $S(e)$, two *borders* \mathcal{A} and \mathcal{B} can be identified from $v_{b,e}$ to r_b along the boundary of the external face of b (see Fig. 3.7.2.a). If b is an elementary block, the two “sides” of the edge $(v_{b,e}, r_b)$ are considered as distinct borders. A vertex w in a block b different from the root r_b is defined to be *externally active for b* if there is a path from w to an ancestor of the currently being processed vertex v , that is only composed by a (possibly empty) sequence of tree edges not belonging to b plus a single back-edge. Fig. 3.7.2.b provides examples of externally active vertices.

Two constraints, called α and β , must be enforced on each path p_e :

- (α) For each block b that contains edges in $S(e)$ the path p_e must contain either the border \mathcal{A} or \mathcal{B} from the entry point $v_{b,e}$ to the root r_b of b .
- (β) A path p_e must not contain a border that has an inner vertex (i.e. a vertex other than $v_{b,e}$ and r_b) that is externally active. See Fig. 3.7.2.a for an example of a forbidden configuration.

Three more constraints, γ , δ , and ϵ , are expressed with respect to each pair of paths p_{e_1} and p_{e_2} traversing the same block b . We say that the border of b used by p_{e_1} *intersects* the one used by p_{e_2} if they share an edge (or if they share the same “side” of the sole edge of the elementary block b). Non-intersecting paths can only share entry or exit vertices (or both) in the blocks they both traverse.

- (γ) If the border of b used by p_{e_1} does not intersect the border of b used by p_{e_2} , the two paths must use non-intersecting borders in all the blocks they both traverse (see Fig. 3.7.2.b for an example).
- (δ) Paths p_{e_1} and p_{e_2} must not use intersecting borders of b if they traverse two other distinct blocks that are externally active. We call a block *externally active* if it contains a (non-root) externally active vertex. See Fig. 3.7.2.c for an example.
- (ϵ) If the border of b used by p_{e_1} does not intersect the border of b used by p_{e_2} and the root r_b of b is different from v , then r_b must not be an outer vertex of $G(v)$. See Fig. 3.7.2.d for an example of a forbidden configuration.

If no set of paths can be found satisfying the above constraints, then G is not planar [BM99].

The Walk-Down Task

The purpose of the walk-down task is to build the embeddings of the blocks of $G(v)$ using the paths selected in the previous walk up. Actually, the information used by the walk-down task is simpler than the paths themselves. It only consists of the set of the borders of the blocks used by the paths (that is, the marked borders) and, for each vertex w that was a walk-up entry point for a block, of two sets of vertices called *active roots* and *non-active roots* of w . The set of the active roots of w contains the roots (at most two) of the externally active blocks traversed by the paths immediately before entering w . The set of the non-active roots of w contains the roots of the non externally active blocks traversed by the paths immediately before entering w .

During the walk-down, some blocks of $G(u)$ will be merged into some blocks of $G(v)$. In order to identify the blocks to be merged, consider two back-edges e_1 and e_2 in $B_{in}(v)$. If the intersection of $S(e_1)$ and $S(e_2)$ is not void, we say that e_1 and e_2 are *interleaving*, and we denote this relation by $e_1 \equiv e_2$. Consider two back-edges e_1 and e_2 in $B_{in}(v)$. If the intersection of $S(e_1)$ and $S(e_2)$ is not void, we say that e_1 and e_2 are *interleaving*, and we denote this relation by $e_1 \equiv e_2$.

By using the fact that the intersection of $S(e_1)$ and $S(e_2)$ (if it is not void) necessarily contains a tree edge incident to v , it is easy to prove that the interleaving relation is an equivalence relation, therefore inducing a partition of $B_{in}(v)$ into equivalence classes $B_{in}(v)/\equiv$. Roughly, the edges of the same equivalence class have the same child of v as an internal vertex of their supports.

Let B be a class of $B_{in}(v)/\equiv$. Denote by Γ_B the embedded blocks of $G(u)$ whose intersection with the support of at least one edge of B is not void. During the walk-down, for each equivalence class B of $B_{in}(v)/\equiv$, the marked vertices of the blocks in Γ_B are visited,

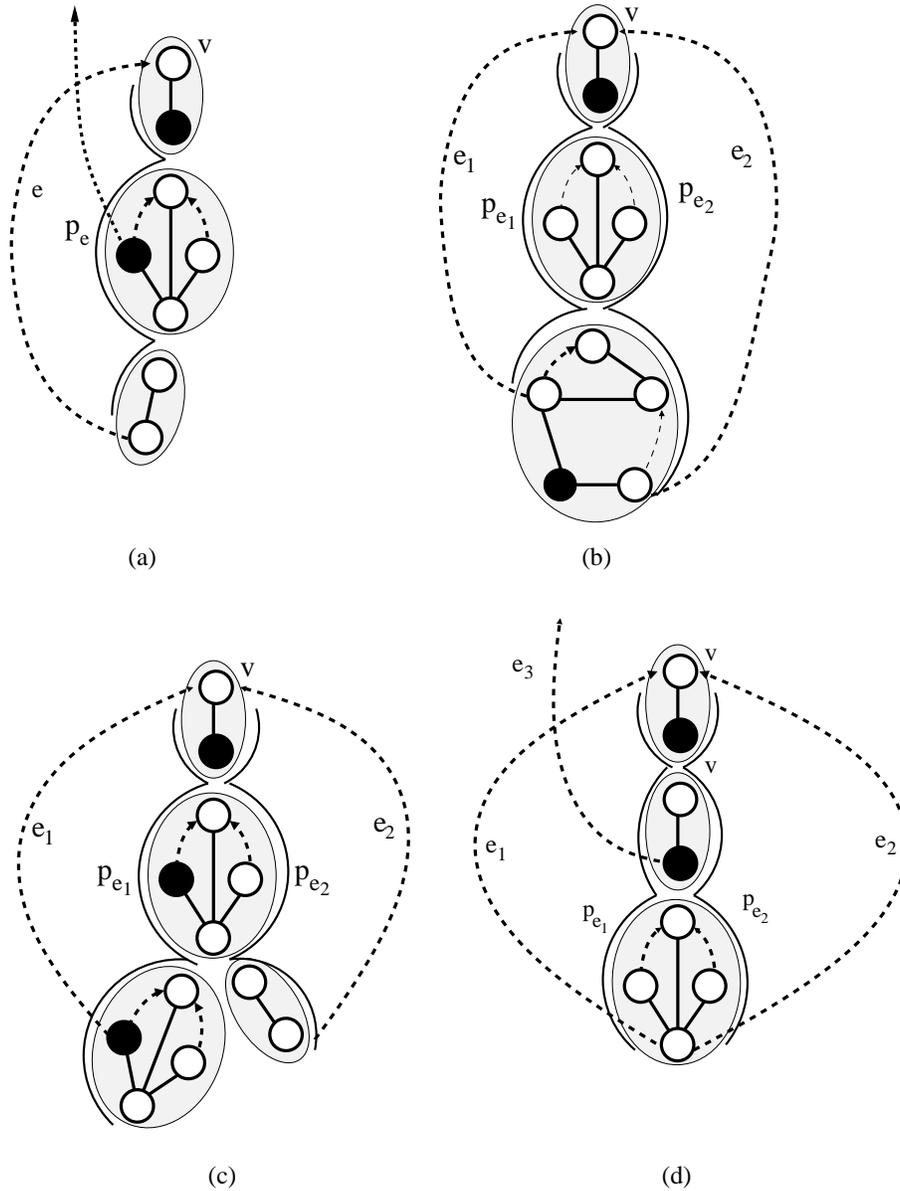


Figure 3.9 Four different configurations for illustrating Constraints β , γ , δ , and ϵ . Edges in $B_{in}(v)$ are drawn with dotted lines, while the paths examined against the constraints are represented with solid lines drawn along the borders of the embedded blocks of $G(u)$. Externally active vertices are drawn black. (a) An example of a path that does not satisfy constraint β . (b), (c) Examples of pairs of paths satisfying constraints γ and δ , respectively. (d) An example of a pair of paths that do not satisfy constraint ϵ .

and a new embedded block b_B of $G(v)$ is obtained by merging the blocks in Γ_B and the back-edges in B (see Fig. 3.7.2.b).

For each class B of $B_{in}(v)/\equiv$, the walk-down process starts on a marked border \mathcal{A} of the elementary block containing v by pushing v onto a stack. When the stack becomes void an

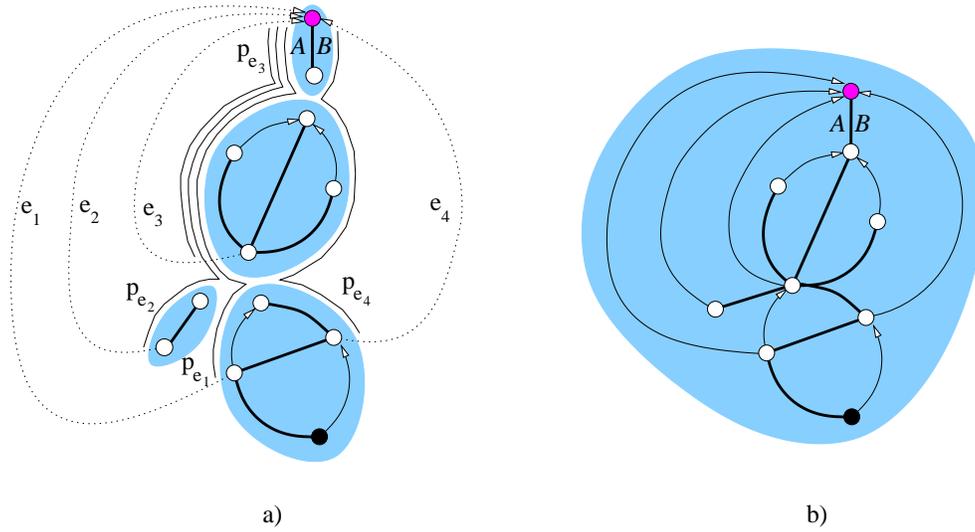


Figure 3.10 (a) The blocks in Γ_B , where B is one class of $B_{in}(v)/\equiv$. For each back-edge e of B , drawn with dotted lines, its corresponding path p_e is drawn with a solid line along the borders of the blocks. The black vertex is an outer vertex of $G(v)$. (b) The embedding of b_B obtained during the walk-down.

analogous process is performed on the opposite border \mathcal{B} , except if v is the DFS tree root. Let w be the current vertex on top of the stack. The back-edge e from w to v (if any) is added to the embedding of b_B in such a way to close the border \mathcal{A} in the internal part of the cycle formed by e and p_e (see Fig. 3.7.2.b). Then, the stack is updated by: (i) removing w ; (ii) pushing onto the stack the next marked vertex x along the border adjacent to w provided that the set of active roots of w is empty; (iii) pushing onto the stack one of the active roots of w (if any); (iv) pushing onto the stack all the non-active roots of w . When the current vertex on top of the stack is found to be the root vertex of some block b , then b is merged into the block b_B , flipping its embedding if necessary so that a marked border of b attaches to the just processed border of b_B . For a while, b_B becomes not biconnected until the descendant endpoint w of a back-edge $e = (w, v)$ is reached. The addition of e will make b_B again biconnected.

The data structure

The embedding construction step works on the blocks, searching paths along the external borders, adding the back-edges and merging the blocks. It is possible to implement the embedding construction step so that it runs in linear time, by using a peculiar data structure that represents the embeddings of the biconnected components. This structure allows the flip of a block in $O(1)$, and the traversal of the external border. An example is shown in figure 3.7.2.

The data structure represent the blocks with an object, (represented by a circle), in correspondence to each vertex, and two objects (the rectangles) for each edge. Note that a cut vertex is represented in each blocks it belongs to. Each object has three links to other objects. The *twin link* connects the two objects relative to the same edge between them. The *neighbors links* connect each object to the two object that were adjacent to it were it was on the external face of the embedding of the block. Given a vertex on the external face of a block, it is possible to visit all the other objects in the external face by

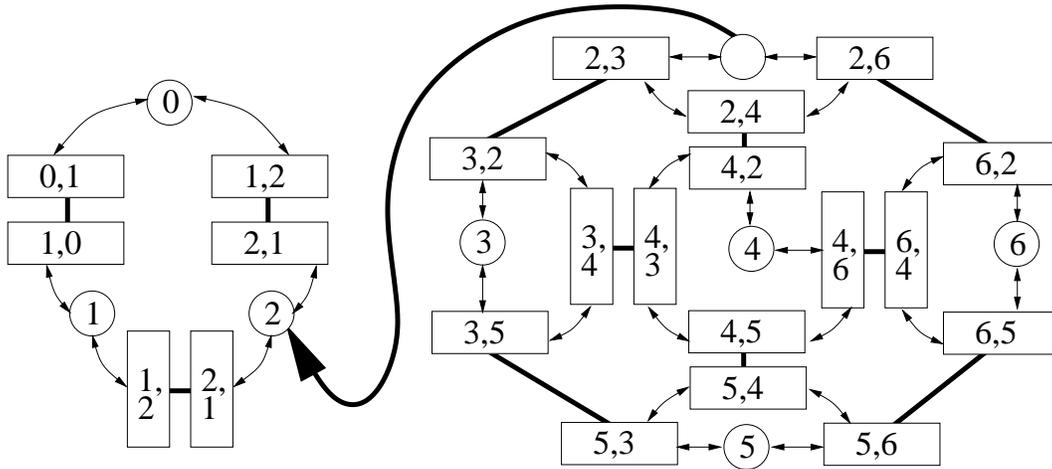


Figure 3.11 The data structure used to represent the embedding of the block

following respectively a neighbors link of the vertex (reaching an edge), the twin link, and then the neighbor link connected to a vertex. Hence, the data structure allows to visit all and only the objects on the external time.

The Embedding reporting step is performed only if the graph is planar. The block data structure represents in an implicit way, at the end of the embedding construction step, a planar embedding of the graph. In this step the embedding is extracted from the data structure and reported to the original graph. The main technique adopted in this step is that each edge of the spanning tree is labeled with 1 or -1 during the embedding construction step; this notation indicates that the child block has been flipped, and so the embedding of the node must be read in reverse order.

The new Boyer and Mirvold Algorithm

Recently, a new version of the algorithm has been presented [BM04]. This new version maintains several features of previous version. The planar embedding is constructed incrementally, starting from a DFT-tree and searching a path along the external border of the block for each back edge. Moreover, the blocks are represented with the same data structure. The new version of the algorithm still visits the vertex in inverse DFI order, performs a Walk-up for each back-edge and a single Walk-down for each vertex; however, the purposes of the two tasks are quite different. Now, the Walk-up collects, for each back-edge to be inserted, informations about the pertinent blocks, but does not establish if a back-edge can be inserted in the embedding. All decisions are made in the Walk-down step; the Walk-down finds a path for each back-edge, and the back edge can be added to the embedding, the Walk-down immediately modifies the data structure.

This new formulation of the algorithm yields a simpler implementation.

Bibliography

- [AH77] K. Appel and W. Haken. Every planar map is four colourable, part I: discharging. *Illinois J. Math.*, 21:429–490, 1977.
- [AHK77] K. Appel, W. Haken, and J. Koch. Every planar map is four colourable, part II: Reducibility. *Illinois J. Math.*, 21:491–567, 1977.
- [BL76] K. Booth and G. Lueker. Testing for the consecutive ones property interval graphs and graph planarity using PQ-tree algorithms. *J. Comput. Syst. Sci.*, 13:335–379, 1976.
- [BM99] John Boyer and Wendy Myrvold. Stop minding your P’s and Q’s: A simplified $O(n)$ planar embedding algorithm. In *10th Annual ACM-SIAM Symposium on Discrete Algorithms*, volume 1027 of *Lecture Notes Comput. Sci.*, pages 140–146. Springer-Verlag, 1999.
- [BM04] John Boyer and Wendy Myrvold. On the cutting edge: Simplified $O(n)$ planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004.
- [CNAO85] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *J. Comput. Syst. Sci.*, 30(1):54–76, 1985.
- [dO] H. de Fraysseix and P. Ossona de Mendez. P.I.G.A.L.E - Public Implementation of a Graph Algorithm Library and Editor. SourceForge project page <http://sourceforge.net/projects/pigale>.
- [dR85] H de Fraysseix and P. Rosenstiehl. A characterization of planar graphs by Trémaux orders. *Combinatorica*, 5(2):127–135, 1985.
- [ET76] S. Even and R. E. Tarjan. Computing an st-numbering. *Theoret. Comput. Sci.*, 2:339–344, 1976.
- [Eve79] S. Even. *Graph Algorithms*. Computer Science Press, Potomac, Maryland, 1979.
- [Har69] F. Harary. *Graph Theory*. Addison-Wesley, Reading, Mass., 1969.
- [Hsu03] Wen-Lian Hsu. An efficient implementation fo the PC-Tree algorithm of Shih and Hsu’s planarity test. Tech. Report, Inst. of Inf. Science, Academia Sinica, 2003.
- [HT65] F. Haray and W. T. Tutte. A dual form of Kuratowski’s theorem. *Canad. Math. Bull.*, 8:17–20, 1965.
- [HT74] J. Hopcroft and R. E. Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974.
- [Kur30] Kazimierz Kuratowski. Sur le problème des courbes gauches en topologie. *Fund. Math.*, 15:271–283, 1930.
- [LEC67] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In *Theory of Graphs: Internat. Symposium (Rome 1966)*, pages 215–232, New York, 1967. Gordon and Breach.
- [Lie01] Annegret Liebers. Planarizing graphs – a survey and annotated bibliography. *Journal of Graph Algorithms and Applications*, 5(1):1–74, 2001.
- [MM96] Kurt Mehlhorn and Petra Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16:233–242, 1996.
- [RSST97] N. Robertson, D.P. Sanders, P.D. Seymour, and R. Thomas. The four color theorem. *J. Combin. Theory Ser. B*, 70:2–4, 1997.

- [SH93] Wei-Kuan Shih and Wen-Lian Hsu. A simple test for planar graphs. In *Int. Workshop on Discrete Math. and Algorithms*, pages 110–122, 1993.
- [SH99] Wei-Kuan Shih and Wen-Lian Hsu. A new planarity test. *Theor. Comp. Sci.*, 223, 1999.
- [Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
- [Tho99] Robin Thomas. Graph planarity and related topics. In Jan Kratochvíl, editor, *Graph Drawing (Proc. GD '99)*, volume 1731 of *Lecture Notes Comput. Sci.*, pages 137–144. Springer-Verlag, 1999.
- [Wag37a] K. Wagner. Über eine eigenschaft der ebenen komplexe. *Math. Ann.*, 114:570–590, 1937.
- [Wag37b] K. Wagner. Über eine erweiterung eines satzes von Kuratowski. *Deutsche Mathematik*, 2:280–285, 1937.

Biconnected component, 2
Block, *see* biconnected component

Component
 biconnected, 2
 connected, 2

Connected components, 2
Cut vertex, 2

Depth first search, 4
 index, 4
 tree, 4

DFS, *see* Depth first search

Drawing, 2
 planar, 2

Embedding, 2

Face, 2
 external, 2

Graph
 biconnected, 2
 connected, 2
 definition, 2
 directed, 2
 planar, 2
 undirected, 2

Palm tree, 4
path
 addition, 3