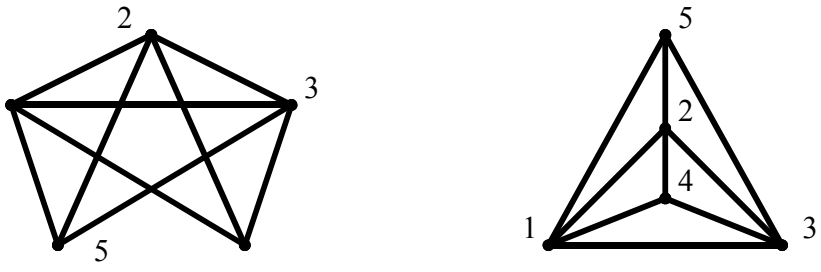
Lecture Notes On Planarity Testing And Construction Of Planar Embedding

1. Introduction

1.1 Introduction - Problem definition

A graph *G* is called *"planar"* if there is a way to draw it in the plane (e.g. on a piece of paper) such that there are no crossings among edges, except of course of the endpoints of edges which may coinside upon common vertices (shared by more than one edge). Given in another way: a graph *G* is planar if a way exists to draw it in the plane such that any of the plane's points are at most occupied by one vertex or one edge that passes through it.

We seek a method which will first test a given graph for the property of planarity and then, if the graph is planar, will produce a representation of the apropriate drawing of the graph. This representation of the drawing which produces no crossings is called a *"planar embedding"* of the graph. It does not describe the length and shape of the edges of the graph nor the position of graph's vertices. Instead, the topology of the graph is represented, which must be necessarily known, in order to produce a drawing. The final drawing of the graph is therefore one of the infinite instantiations of the topology that is described in the graph's planar embedding, depending on the length, shape and position of edges and vertices respectively. The method we seek will not produce the actual planar drawing of the graph, but will compute the planar embedding. The way to use the planar embedding in order to produce an actual drawing with certain appearance and desired properties is another distinct problem, which we will not study here and highly depends on the application that uses the drawing. In figure 1.1, we depict an example of a graph subjected to planarity testing and a possible planar drawing of the graph, in order to show that the problem is not as easy as it seems from a first glance. Still, we remind that we do not seek the actual drawing, but only its topology.



Input Graph Planar drawing

Figure 1.1: Given graph and constructed planar drawing

But how can we represent the topology of a graph? Well, the topology of a graph changes if the arrangement of neighbours changes for at least one node of the graph. More specifically, if the order of appearance of edges incident to a vertex is different, then the arrangement of its neighbouring vertices is different, which correspond to a different topology. Therefore, if we represent the graph with an adjacency list, then the topology can be represented with an appropriate ordering of the edges incident to any vertex (clockwise or counterclockwise) such that the order of the edges in the adjacency list of each vertex reflects the order of the appearance of the edges if someone would look at a planar drawing of the graph.

We remind that the adjacency list is a data structure that uses an array of pointers, having one slot for each vertex of the graph. Each pointer points to a list of nodes, each one corresponding to a vertex of the graph that is linked through an edge to the specific vertex. Therefore, the nodes in the list are the neighbours of the specific vertex and each node indicates and requires the existence of an edge. Arranging the order of appearance of the neighbouring vertices in the list changes the order of appearance of the edges that link the specific vertex with its neighbours.

1.2 Applications

The need for a test of planarity for a given graph and the construction of a planar embedding arises in many applications. A classical example is in the area of VLSI design, where a designed circuit, which can be represented as a graph having edges that correspond to wires, must be planar, because it will be printed on a surface and a crossing would mean that two paths would communicate, creating side-effects on the outcome of the logical operations of the circuit. The construction of the embedding reveals the structure of the printed circuit, that will implement their designed logic functions, while a planar drawing, under specific constrains would show an image of the circuit itself.

Other applications include road networks and other fields that have to do with structures represented as graphs which operate or must be embedded to planar surfaces. Appart from such applications, there are other cases where the planarity of a graph can be exploited, since planar graphs have certain properties that simplify the solution of some problems. These properties are often outcomes of the fact that a planar graph is a sparse graph – Euler's formula constrains the edges of a planar graph to be below the number of vertices of the graph multiplied by 3, which means that a planar graph's edges can be examined one by one in linear time –. Therefore, an algorithm to detect planarity is really needed in some cases.

Finally, if we consider the computation of a planar embedding for a given graph as a first step towards the construction of a planar drawing, then there are even more applications, based on the fact that a drawing of a graph in a way that eliminates crossings, yields simpler and thus intuitive and correctly interpreted schemas.

1.3 Short historical information

Much of the work in graph theory is motivated and directed to the problem of planarity testing and construction of planar embeddings. Inversely, much of the development in graph theory is due to the study of planarity testing.

Auslander and Parter [AP61], in 1961 and Goldstein in 1963 presented a first solution to the planarity-testing problem. Demoucron, Malgrance and Pertuiset also presented an algorithm in 1964 [DMP64]. Lempel, Even and Ceberbaum, in 1967, gave a solution to the problem in time $O(n^3)$ [LEC67]. Their algorithm used vertex addition and st-numbering.

In 1974, a significant breakthrough was achieved when Hopcroft and Tarjan, using path addition and depth first search, presented an algorithm to test planarity in linear time [HT74]. Still, this algorithm only returns whether a graph is planar, but it does not actually find the planar

embedding, although the authors showed that it could be constructed. Another disadvantage of this algorithm, like all other algorithms created later on to solve the problem in linear time, is that it is very complicated either to explain or to implement.

In 1976, two independent results showed together that the algorithm by Lempel, Even and Cederbaum mentioned above actually also can be implemented in O(n) time. First, Even and Tarjan [ET76] showed that an st-ordering of a bi-connected graph can be found in linear time. Then, Booth and Lueker introduced the PQ-tree [BL76], and showed that it can be used to test the consecutiveness of predecessors in overall linear time. Still, this algorithm does not compute the planar embedding of the graph, but in 1985, Chiba et al. [CNAO85] showed how to use the algorithm to find the embedding in linear time.

In 1992, Hsu and Shih [SH92] gave an entirely new approach to planarity testing, based on doing a depth-first search with appropriate bookkeeping. This approach was followed and clarified independently by Boyer and Myrvold [BM99], in 1999.

In 1996, Mehlhorn and Mutzel implemented Hopcroft and Tarjan's algorithm as part of the LEDA program package, and in the process, clarified the algorithm and explained how to find the planar embedding in linear time as well [MM96]. Also in 1996, Di Battista and Tamassia developed an on-line planarity-testing algorithm [BT96]. Here, the question is to decide whether the current graph is planar, under a sequence of changes to the graph that may or may not destroy or add planarity.

1.4 Initial considerations

We need to test if a graph is planar and if it is we want to construct its planar embedding. We need a solution that will work for any kind of graph.

If a graph, whose planarity is being tested, has more than one connected components, then it is easy to show that the graph is planar iff each of its components is planar. Intuitively, this means that if a graph has parts (components) that do not communicate with other parts, then these can be drawn separately and independently from the others and as a result their property to be planar or not is derived exclusively from their own internal structure. Therefore, we can check if each of these parts is planar and then draw a conclusion for the total graph. Practically, this means that we only need to construct a method that will work exclusively on connected graphs.

Moreover, if our connected component, whose planarity we check, has one or more cut-vertices, then we can remove these vertices and check each of the produced sub-graphs individually. We remind that a cut-vertex is a vertex that if removed, then the graph is separated into two connected components. The underlying idea is the same as before: We can test and draw the sub-graphs independently and them bring them close to form the total/initial graph (so as two vertices, one belonging to the first sub-graph and one belonging to the second, coincide shaping the former cut-vertex). As a result, we only need to deal with graphs of higher density, those that are bi-connected. Bi-connected graphs do not have cut-vertices. This means that for each vertex of the graph, there is at least one cycle that contains it, so as a removal of whatsoever vertex, may cut/break one path to some vertex but will leave at least a second path available. The graph in Figure 1.2 has two connected components, C_1 and C_2 respectively. C_1 is bi-connected, but C_2 is not, having the cut-vertex v, which divides C_2 into two bi-connected parts C_{2a} and C_{2b}.

Drawing a graph with directed edges is not different from the case of an undirected graph, concerning the property of planarity, since it is the existence of an edge, which may create a crossing and not its direction. Therefore, we can consider undirected graphs being subjects for planarity testing. Of course, the information of direction, if existent, can be kept and added later on the graph drawing.

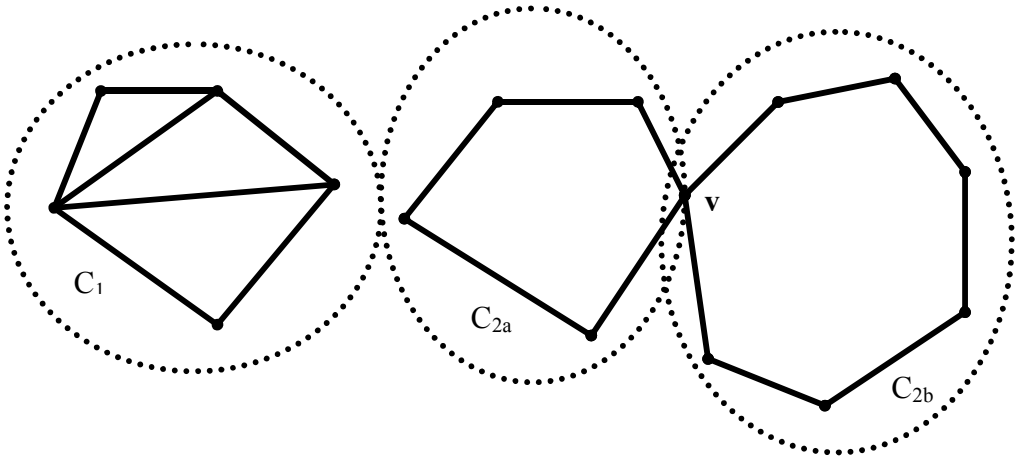


Figure 1.2: connected and bi-connected components, cut-vertex

As a result from the above considerations, when performing a Depth First Search on a graph *G* being tested for planarity, the resulting graph D_G (DFS tree + back-edges) will have only two types of edges: tree-edges (edge set *T*) and back-edges (edge set *B* - the set of all non-tree edges). This is an outcome of the property of the graph being undirected. Furthermore, from any leaf of the DFS tree there is at least one back edge to a tree vertex. Also, there must be at least one back-edge that will lead to the root of the tree, because in a different case, the root and maybe a part of the tree (and the corresponding part of the graph in which the DFS is performed) could be disconnected by a removal of a cut-vertex. This cannot happen since we have a graph, which is bi-connected.

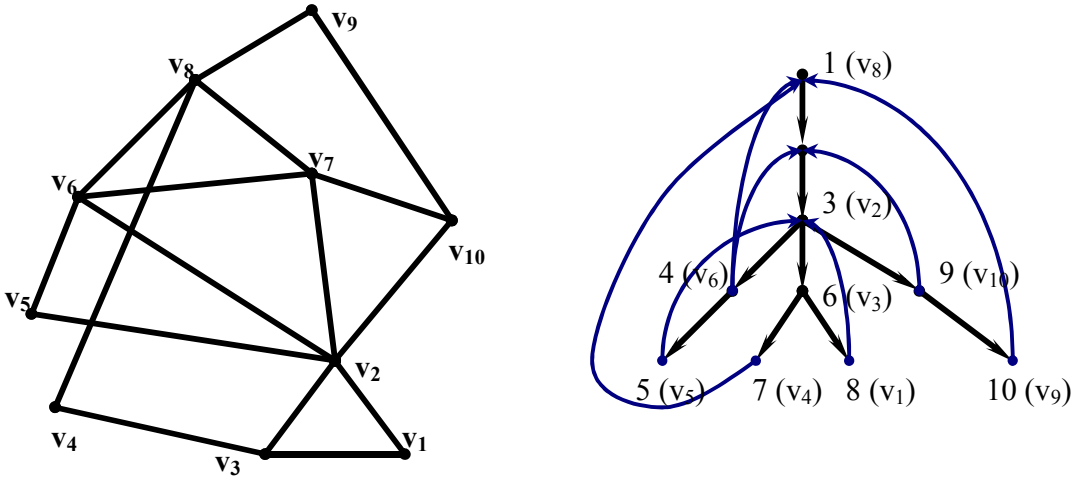


Figure 1.3: Given initial graph G and corresponding DFS D_G

From now on, we will identify the nodes of the DFS graph D_G with their DFS number, therefore when we refer to vertex *x*, we mean the vertex whose DFS-number is *x*. Furthermore, we direct tree-edges from lower to higher DFS-number vertices, and back-edges from higher to lower DFS-numbers. In figure 1.3, an example of an input graph is depicted along with its corresponding DFS graph, D_G. The DFS-numbers of D_G vertices and the number of initial graph's vertex v_i to which they correspond are shown on D_G. All the information we need in order to check graph *G*'s

property of planarity is contained on D_G and this is the input to the algorithm of Hopcroft and Tarjan, which we will present and analyze in the sequel.

2. The algorithm of Hopcroft and Tarjan

2.1 Fundamental idea

The algorithm of Hopcroft and Tarjan is heavily based on the tree that is produced by Depth First Search. It uses the DFS tree first to find a set of edges that form a cycle inside the graph (which is always existent in a bi-connected graph) and then it uses the tree to retrieve components (sub-graphs) of the graph that are attached to this cycle (through edges). Each of these components is tested for planarity and if it is planar its embedding is constructed. After a new component is tested, its representation is added to the existing embedding of the graph and this way the embedding increases until it represents the whole graph. If two components intersect with each other, then one of them is placed inside the cycle (to whom both components are attached) and the other is placed outside, otherwise both components are placed inside or outside the cycle. If one component intersects at the same time with at least two already tested and represented components, one being inside the cycle and the other outside the cycle, this means that this part of the graph cannot be drawn either inside the cycle or outside without intersecting with the other parts of the graph and therefore the graph is not planar.

The order to visit the components of the graph attached to the initial chosen cycle is discovered after an appropriate reordering of the adjacency lists of the DFS vertices, which is done before the main work of the algorithm. The order of visiting the components along with their definition is what makes the algorithm of Hopcroft and Tarjan different from other planarity testing algorithms and above all it ensures that the time of producing the embedding or answering that the graph is not planar is linear.

2.2 Retrieving the components of the graph

Let's consider an edge e of the DFS graph D_G. If the edge e is directed from the vertex with DFS-number x to the vertex with DFS-number y then we can write $e=(x,y)$. This edge of course corresponds to an edge of the initial graph G. If $x < y$ then this is a tree-edge, otherwise ($x > y$) its a back-edge, related to D_G, and it "closes" a cycle of edges: (y,w_1), (w_1,w_2), ..., (w_i,y), (x,y), with all other edges, except (x,y) being tree-edges. In the case where $e=(x,y)$ is a tree-edge, there is still a cycle in which edge e is involved/participating and maybe more than one. To be exact, for each back-edge that exists in the DFS sub-tree below edge e, a different cycle is defined in which e participates. Each cycle has two parts: The first part of the cycle, called *"spine"*, starts from vertex y - the endpoint of edge e -, follows a path of tree edges and ends in vertex a of some back-edge (a,b). If edge e were a back-edge, then the spine would be empty. The second part starts from the vertex b, which has of course lower DFS-number than x and is closer to the root of D_G - or the root itself - and ends to vertex x, following the corresponding tree path. This part is called, the *"stem"*. In the notion of *cycle* $C(e)$ for a given tree-edge $e=(x,y)$, from all possible cycles, we include only those that use a back-edge (a,b), whose endpoint b has the lowest DFS-number among the endpoints of the back-edges in the sub-tree below edge e. This definition has certain properties useful when checking a graph's planarity. Again there can be more than one paths that can lead to the same lowest DFS-number vertex b, through using different back-edges (who have the same endpoint b). Therefore there are more than one possible cycles for a given tree-edge. In figure 2.2 an example is depicted. By appropriately reordering the adjacency list of each vertex of the DFS graph we can easily construct the cycle for a given edge e. Then, the spine of the cycle for the tree-edge $e = (x,y)$, is constructed by starting from y and always taking the first edge out of

each node, until a back-edge is encountered. Thus, for each edge e of D_G, we have defined a cycle $C(e)$ having two parts: the stem and the spine. In figure 2.1, a DFS graph is shown along with two examples of cycles for edges $e_1 = (2,3)$ and $e_3 = (9,1)$ respectively.

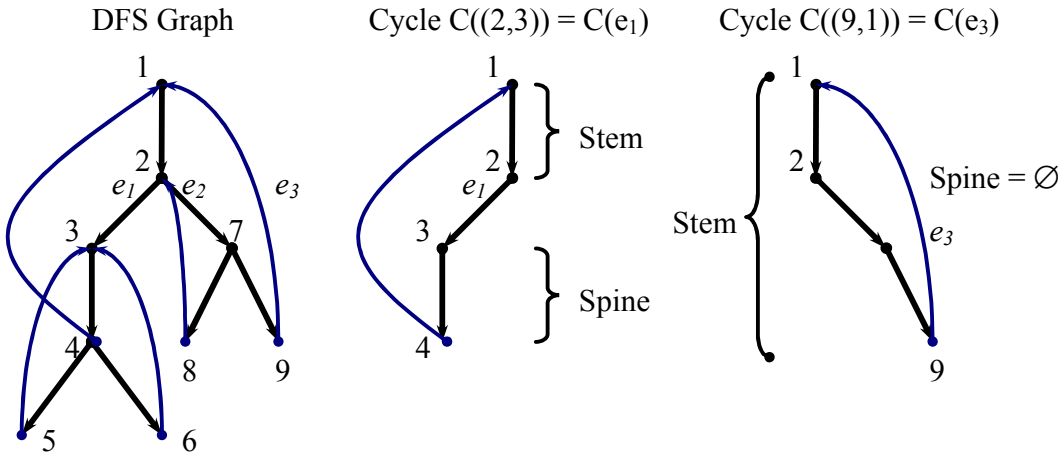


Figure 2.1: Examples of cycles for an edge *(e_1)* and a back-edge *(e_3)*.

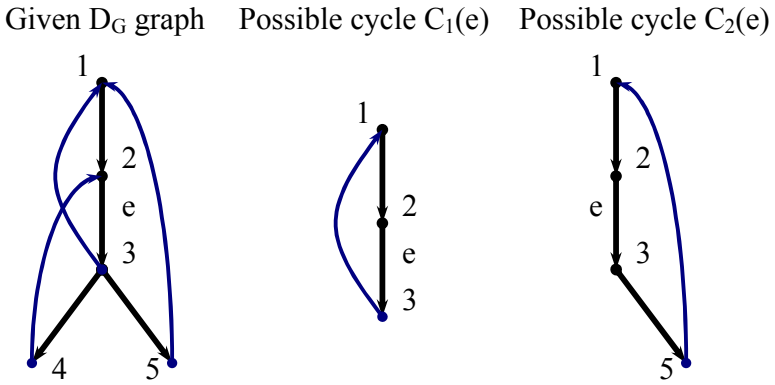


Figure 2.2: 2 possible cycles for tree-edge e.

We can now define the *segment* $S(e)$ for an edge e as the union of the cycle $C(e)$ along with the rest of the sub-graph of D_G below edge e - tree-edges and back-edges -. Of course, if the edge e is a back-edge then $S(e)$ matches with $C(e)$. There is only one possible segment for a tree-edge e, since it includes the whole sub-graph below edge e and not only a specific path. As someone can easily note, $S((1,2))$ i.e. the segment of edge (1,2), corresponds to the whole graph, since it includes all tree edges and back-edges below the root of D_G. Therefore, testing the initial graph for planarity equals testing $S(1,2)$ for planarity (actually, $S(1,2)$ must have yet another property except for being planar, but the details will be explained later on).

As we said before, the main idea of the Hopcroft and Tarjan algorithm is to find a cycle C_0 of edges and all the connected components of the graph that are created if C_0 is removed from the graph. Then, test the components for planarity separately and check if can be drawn altogether, along with the cycle, (some components placed inside the cycle and others outside) without crossings. Each component can be tested separately for planarity, because it does not communicate with the others apart from that they may have common endpoints upon the cycle C_0. As a result, we can check if the component itself is planar and then check if it can be drawn

together with the rest of the components, given the topology / (relation & combination) of all components' endpoints. The same procedure can be applied to check if a component is planar and this is done through recursive calls of the algorithm.

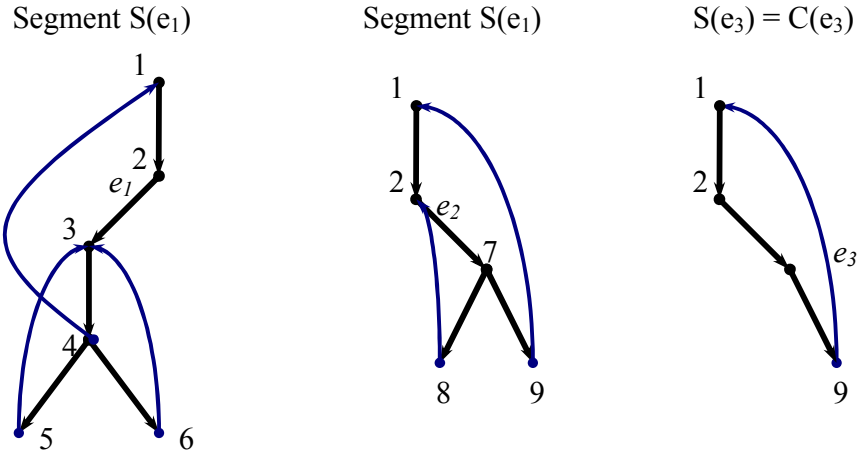


Figure 2.3: Examples of segments for edges (e_1, e_2) and back-edge (e_3)

Lets see how we can discover the connected components, related to a cycle C_0=$C(e_0)$, computed for some edge e_0:

An edge e=(x,y) is said to *"emanate"* from C_0, if x lies on the spine of C_0 but e does not belong to C_0. First of all, e is directed from x to y. This means that it "leaves" the cycle either going towards the inside face or the outside face of the cycle. Edge e belongs to one of the components we seek, because it does not belong to the cycle. At the same time, since any connected component of the graph, related to the cycle C_0, cannot have edges that belong to both the interior and the exterior face of C_0, then x is part of the border of a (interior or exterior) connected component related to C_0, i.e. one of its endpoints.

A component, related to a cycle C_0 of the graph G, is always starting from a tree-edge e_i that has its start-point on the cycle. Then, it includes the whole sub-tree of D_G that is descendant from e_i = (x_i,y_i) and all back-edges that start from a vertex in the specific sub-tree. It "touches" the cycle exclusively at x_i and at one or more end-points of back-edges that belong to the component. There is at least one back-edge whose end-point belongs to the cycle C_0, otherwise the removal of x_i would disconnect the component and thus the whole graph would be disconnected - x_i would be a cut-vertex - which is not valid for our bi-connected graph. All back-edges that are included in the component are directed, either towards vertices that are parts of the tree-edges of the component or towards vertices that belong to the cycle C_0, given the way Depth First Search works for undirected graphs. Thus, there can be no "communication" among components through back-edges - otherwise they would not be independent and by definition they would not be different components - (Because through this supposed back-edge, DFS would explore one component after the other rendering the second one sub-tree of the first one). At the same time, there cannot be more than one D_G tree-edges emanating from the cycle and being part of the same component, because that would mean that those two tree-edges that belong to different branches of the tree would have a common descendant, which cannot happen for a tree structure.

We therefore realize that each edge ei that emanates from C_0 introduces a new - different component. Furthermore, the exploration of the segment $S(e)$ yields the whole component and only it. To be exact, the definition of a segment matches the definition of a component (based on

graph D_G) plus the part of C_0 that intervenes between the lowest and highest "touch" vertices [1] of the component and the cycle. The reason we add some part of the cycle to what we call a component and thus in the definition of the notion of segment is that the (independent) components are created by removing the cycle C_0, but the initial graph contains the cycle as well and not only the components and therefore we must add the part of the cycle that is adjacent to each component and check them together for the property of planarity.

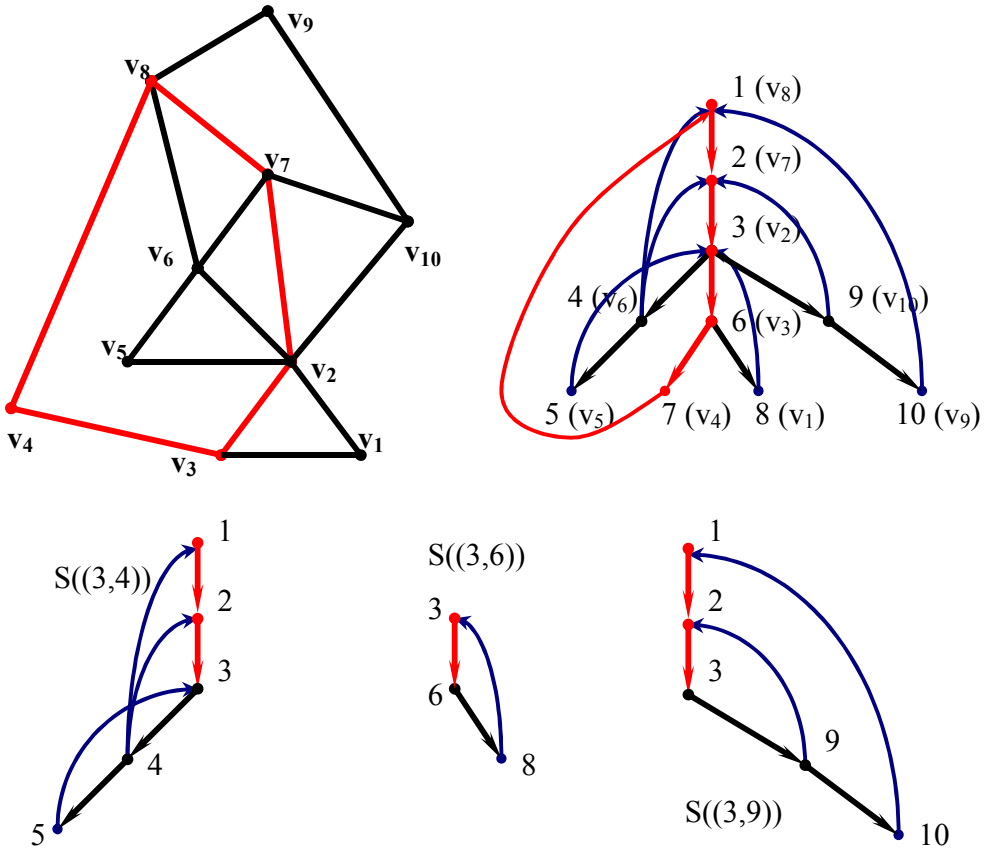


Figure 2.4: A graph with a cycle and components. Its DFS graph, the cycle and components.

In figure 2.4, a planar embedding of the graph of the figure 1.3 is depicted. Then, the DFS graph D_G produced for the graph is shown. The red lines – edges in both the embedding and D_G form the cycle C_0, which defines the components of the graph, tested separately for planarity. The edges that emanate from $C_0 = C((1,2))$ are (3,4), (3,6) and (3,9). These edges correspond to the depicted segments $S((3,4))$, $S((3,6))$, $S((3,9))$ respectively, which correspond to the components of the graph, related to cycle C_0.

---

[1] Note: These common vertices of "touch" between the cycle and the component are called *"attachments"*. Thus For each component / segment, the attachments are: the start-point and end-point of a back-edge, if the component is that specific back-edge alone. Otherwise: the start-point of the tree-edge that emanates from the cycle (from which the exploration of the segment begins) and all the end-points of back-edges that return to a vertex of the cycle. The set of attachments for a segment $S(e)$ is denoted $A(e)$. In figure 2.4, attachments of $S((3,4))$ are the vertices {1,2,3}. Attachments of $S((3,6))$ are {3,6}. Attachments of $S((3,9))$ are {1,2,3}.

If e_1, ..., e_m are the edges emanating from C_0 then $S(e_0) = C_0 + S(e_1) + ... + S(e_m)$, i.e. $S(e_0)$ is the union of the cycle C and the segments $S(e_1)$, ..., $S(e_m)$. This means that in order to test a segment / component for planarity, we can find a cycle inside it and test the planarity of all its internal components (and their combination) that are created by the removal of the cycle. Again, in order to test an internal component for planarity, we can find a cycle inside it ... the components inside the internal component ... etc. This is a clear definition of a recursion that finally reaches to sub-components, which are minimal and obviously declared planar or not planar.

Let's consider that we discover a component through an edge e_i that emanates from a cycle. Thus, we recursively call the procedure that tests for planarity for the segment $S(e_i)$. Then, the cycle $C(e_i)$ is computed and all edges ei' that emanate from it are explored as internal components. Since, cycle $C(e_i)$ surely includes vertices of the initial cycle that are above the start-point of e_i (a consequence of the fact that the graph is bi-connected), there is a possibility that other parts of the graph would be examined, starting from those vertices, that do not belong to the particular component / segment that we test and may even lie in the opposite face of the initial cycle. This is a possible scenario and is the reason that when defining the notion of emanation, it was carefully stated that an edge ei emanates from a cycle $C(e_0)$, only if its start-point belongs to the spine of the cycle, which is the part of the cycle below edge e_0, for which it was computed. This way, the exploration of a component is confined to the part of the graph that belongs to it. Thus we know that when searching the segment $S(e_i)$ the algorithm discovers a whole component and only it.

2.3 Interlacement of segments

Let $S(e_1)$ and $S(e_2)$ be any two (planar) segments explored through edges e_1, e_2 that emanate from a cycle C. By definition, they do not share any edges, except for (possibly) some common edges that belong to the cycle C and are part of the stems of cycles $C(e_1)$ and $C(e_2)$. There is a case, where the two segments cannot be drawn in the same side (face) of the cycle C, since the edges that end or begin by vertices that are attachments of the segments are in such an order (topological relation) that crossings are created independently of the way we use to draw them together. When this happens, we say that the two segments *"interlace"* and we draw them in different sides of the cycle C. Examining the attachments of the two segments is enough in order to discover if the segments interlace or not. Specifically, there are two cases two segments interlace:

1. If they have attachments x, z and y, u respectively, where $x < y < z < u$
2. If they have three or more common attachments.

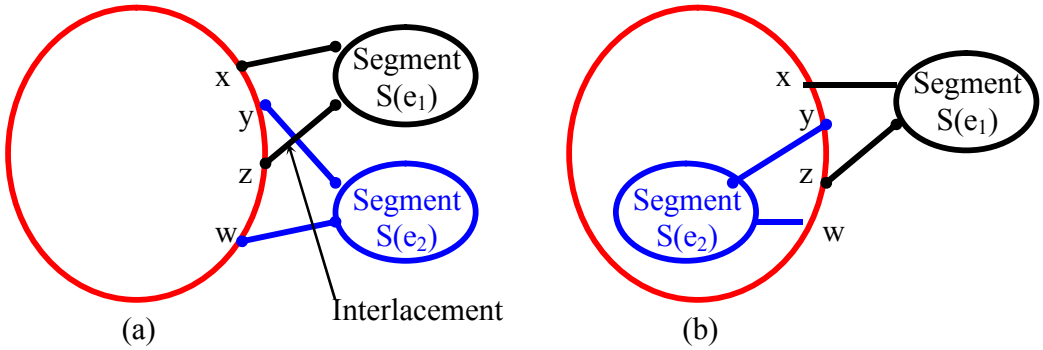


Figure 2.5: Case where segments' attachments are in mixed order

If case (1) happens – see figure 2.5(a) – then let the face *F* be created when drawing the first segment with attachments *x*, *z*. The attachment y is between the attachments *x* and *z* and in order to draw the segments in the same side of the cycle *C*, the only one way to place the second segment so that it has access to its attachment *y* is to draw it inside the face *F*. But if the segment is placed inside face *F*, then it is impossible to have access to its other attachment *u*, which is outside the face *F*, without a crossing. Therefore, the segments interlace, as it is inevitable to have a crossing, if we draw both segments in the same side of the cycle. In figure 2.5(b) the interlacement is avoided through placement of the two segments in different sides of the cycle.

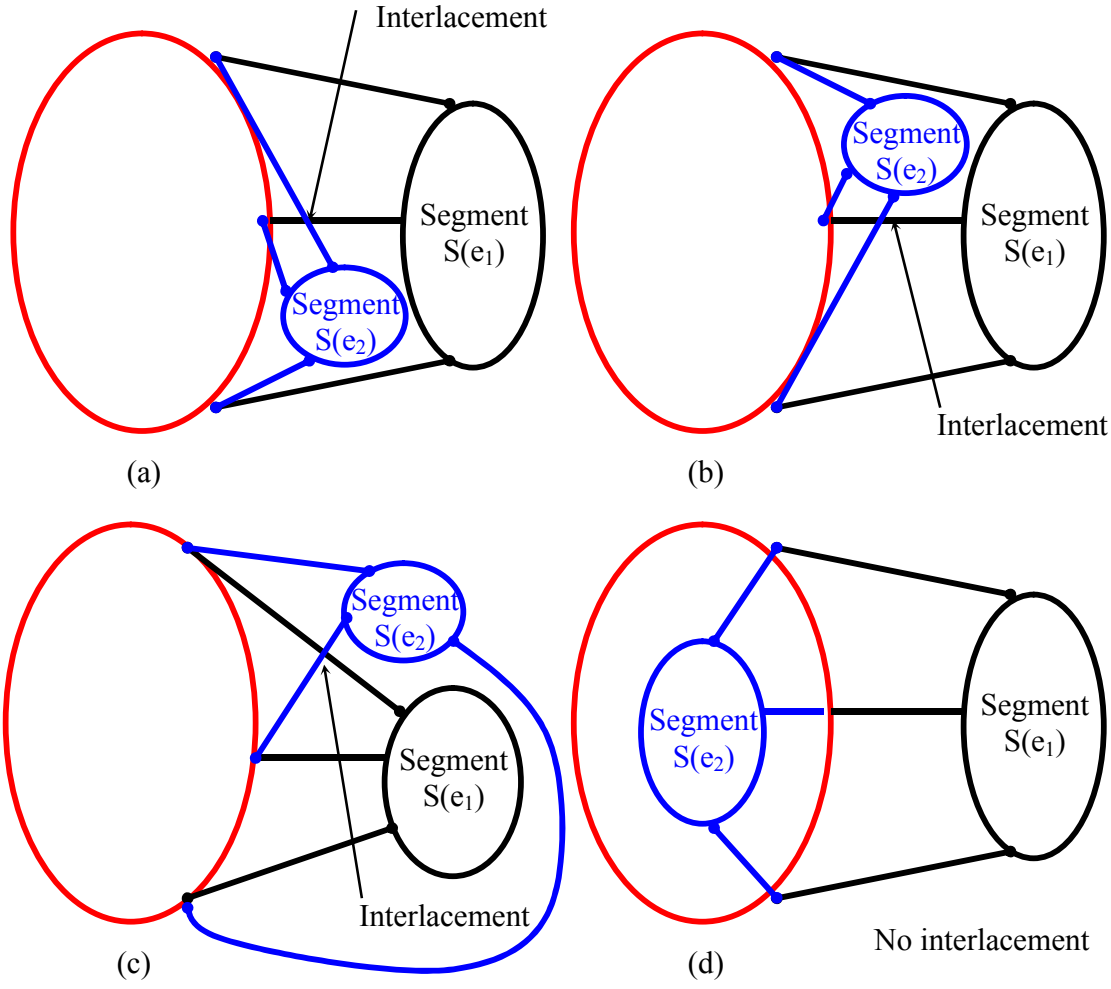


Figure 2.6: We have 3 cases: segment 2 placed in Faces *F1*, *F2*, *F-outer* correspondingly.

If case (2) happens and there are three common attachments – see Figure 2.6 – then it is obvious that drawing the second segment in any of the possible faces (3 cases depicted in figure 2.6 (a), (b) and (c) respectively) that are created by the drawing of the first segment in the specific side of the cycle produces a crossing. Therefore, the two segments interlace and must be drawn in different sides of the cycle – see Figure 2.6(d) –. The same holds for any number of common attachments greater than three, necessarily increasing the number of crossings. On the other hand, if there are only two common attachments, then the second segment can be drawn inside the face

F that is created by the cycle C, two edges of the first segment that and the rest of the first segment – see Figure 2.7 – and thus the two segments do not interlace.

Let's see now how we can check if all the (planar) segments $S(e_i)$ that emanate from a cycle C can be combined without crossings: Having examined two segments and placed them in appropriate sides of the cycle C, we examine the rest of the segments one by one. If a new segment does not interlace with the segments placed inside the cycle C, it is added inside the cycle. If it does interlace, then the case of placing the segment outside the cycle is examined. If one or more interlacements of the segment with the other segments that are placed outside the cycle come along, then the segment cannot be placed either inside or outside the cycle without crossings; thus we declare the graph not planar. We can express the stated requirement more formally: If we create a graph, having as nodes the segments $S(e_i)$ and two nodes are connected with an edge iff their corresponding segments interlace, then the graph constructed this way must be bipartite, in order that the segments can be drawn altogether along with the cycle C, without crossings. The graph constructed this way is called the *"interlacement graph"* $IG(C)$.

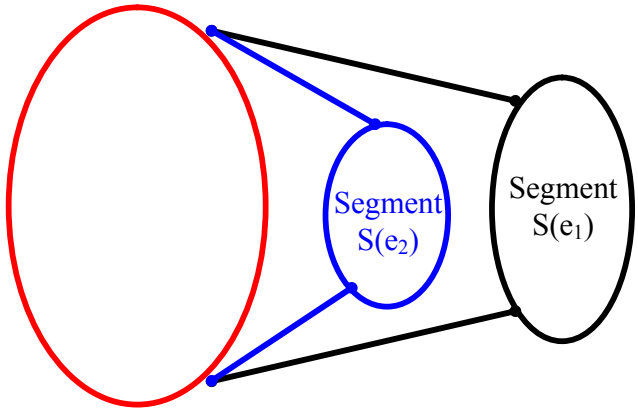


Figure 2.7 When only two common attachments, interlacement is avoided

But there is a case, where a segment $S(e_i)$ (= component along with the part of the cycle that lies between its attachments) even though is planar and does not interlace with other segments, it cannot be placed neither inside nor outside the cycle C – based on which, the segments are considered – and this fact may not have to do with the existence of other segments at all. In such a case, it is the segment's own structure that causes the side-effect, because it may be required that the edges of the cycle C that belong to the stem of cycle $C(e_i)$ – and thus are included in the segment $S(e_i)$ – are placed in the interior face of the embedding of $S(e_i)$ (in-between sub-parts of the segment) and that there cannot be otherwise if we want the embedding to be planar. Therefore, in order to construct a planar embedding of the segment $S(e_i)$, we are forced to encircle the stem of $C(e_i)$, which is part of the cycle C. Thus, we are not able to embed the segment neither inside nor outside the cycle C and this renders the whole graph not planar. An example is depicted in figure 2.8. A given graph is shown and its corresponding DFS graph. The red lines in the initial and DFS graph represent the cycle C, from which the segments are considered (and emanate). The segment $S(e_i=(4,5))$ is an example of a segment that belongs to the DFS graph, it is planar but it is not strongly planar. The segment itself is planar, as it is shown in the bottom of the figure 2.8, but it cannot be embedded in any side of the cycle C, since its embedding surrounds the part of C that belongs to the segment, namely the edges (2,3) and (3,4). This renders the whole graph not planar, because the part of C, which is surrounded by the embedding of the segment, cannot communicate with the rest of the cycle without a crossing.

This effect is not detected if we check only for planarity of segments and interlacement among them. Instead, we need to reassure as well that there is an embedding of each segment where its stem is on the border of the outer face, which renders the segment embeddable in one side of the cycle. A segment having this property is called *"strongly planar"*. Therefore, in order to check that a segment $S(e)$ is strongly planar, we must ensure that it is planar and at the same time that the stem of the cycle $C(e)$, which is part of $S(e)$, has no sub-segments attached to it drawn in both sides, but only in one of the sides, let it be the left one, without loss of generality. More formally, it has been proven that a segment $S(e)$ is strongly planar if $S(e_i)$ is strongly planar for every e_i emanating from $C(e)$ and there is a partition {L, R} of the vertex set of $IG(C(e))$ such that no segment in L interlace with a segment in R and such that $A(e_i) \cap \{w_1, \ldots w_{r-1}\} = \varnothing$ for any segment $S(e_i) \in R$ where $w_0, w_1, \ldots w_{r-1}, w_r$ is the stem of cycle $C(e)$ – which is a clearly recursive test –.

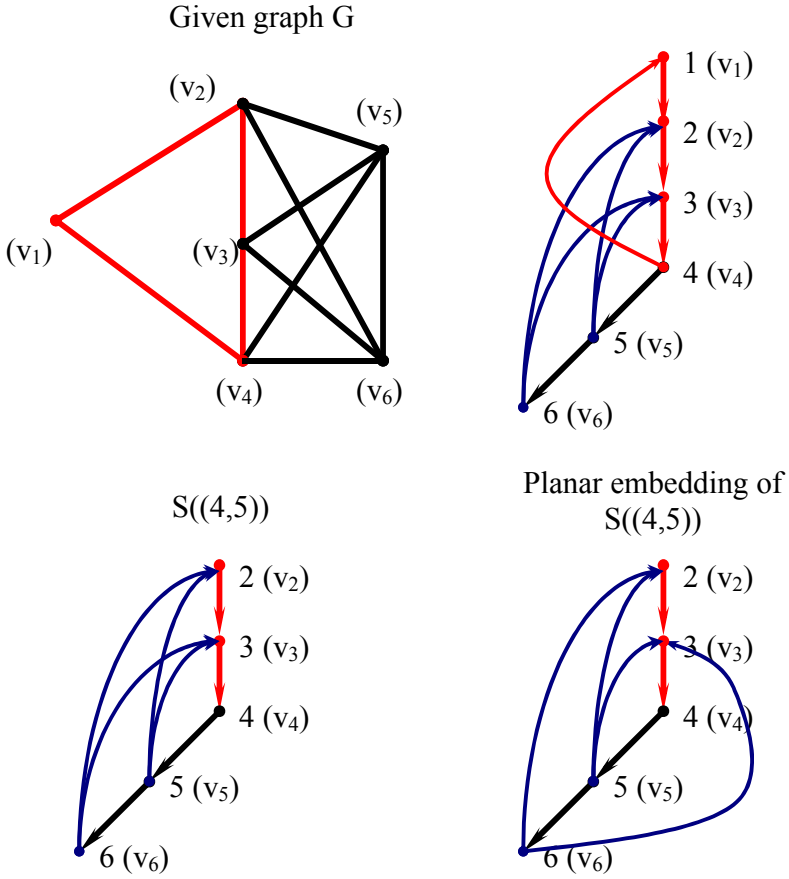


Figure 2.8: Case of a segment that is planar but not strongly planar and thus can be embedded neither inside nor outside the cycle.

2.4 The algorithm

We can now examine the algorithm itself. It tests a given segment $S(e_0)$ for strong planarity. This is done recursively, as the above discussion – sections 2.2 and 2.3 – has suggested and indicated. The input to the algorithm is the DFS graph itself and an edge e_0 in it. The corresponding segment $S(e_0)$ will be tested for strong planarity and the ordered list of attachments of the segment will be

returned, in order to be used (in a higher level) to judge if the segment interlaces with other segments.

---

check_strong_planarity
Input: A DFS graph D_G and an edge e_0 in D_G.
Output: Declaration of the segment $S(e_0)$ as "strongly planar" or "not strongly planar" and the set of its attachments

(1) **procedure** check_strong_planarity (e_0: edge)

(2) **co** Tests whether segment $S(e_0)$, $e_0=(x, y)$ is planar. If so, it returns the ordered (according to DFS-number) list of attachments of $S(e_0)$ excluding x **oc**

(3) Find the spine of cycle $C(e_0)$ by starting in node y and always taking the first edge on every adjacency list until a back-edge is encountered. This back-edge leads to node w_0 = $lowpt[y]$. Let $w_0, ..., w_r$ be the tree path from node w_0 to $x = w_r$ and let $w_{r+1} = y, ..., w_k$ be the spine constructed above.

(4) **for** j **from** k **downto** $r + 1$
(5) **do for all** edges e' (except the first) emanating from w_j
(6) $A(e')$ ¬ check_strong_planarity(e')
(7) **if** segment $S(e')$ can be combined with the already examined part of the graph (i.e. the rest of the segments) without crossings.
(8) merge existent embedding with the embedding of $S(e')$
(9) **elseif**
(10) declare graph "not planar"
(11) stop
(12) **fi**
(13) **od**

(14) **co** if control reaches here, then interlacement graph is bipartite **oc**

(15) $L \leftarrow \varnothing$
(16) check whether segment $S(e_0)$ is not strongly planar and update list L of attachments of segment $S(e_0)$, given $A(e')$
(17) **if** $S(e_0)$ not strongly planar
(18) declare segment "not strongly planar"
(19) stop
(20) **fi**

(21) return L
(22) **end**

The lowest attachment of a segment $S(e_0=(x,y))$, i.e. the endpoint of the back-edge that has the lowest DFS-number of all endpoints of back-edges that belong in the DFS sub-graph below e_0, is denoted as $lowpt[y]$ (since it is the lowest point reachable from y). The lowest point reachable from y, excluding $lowpt[y]$ is denoted $lowpt2[y]$. By computing these two numbers for each vertex of the DFS-graph, a reordering of the adjacency lists of the vertices can be done, in

order to render certain procedures, like the computation of cycles, easier. This reordering is done before calling the procedure check_strong_planarity for the first time and is achieved in linear time, using bucket sorting. The reordering, sorts the outgoing edges of a vertex v as follows: An edge (v,w) is before edge (v,w') if $lowpt[w] < lowpt[w']$ or if $lowpt[w] = lowpt[w']$ and segment $S(v,w)$ has two attachments and $S(v,w')$ has three or more attachments. In all other cases the order is irrelevant. This kind of reordering guarantees that by taking the DFS tree-path from e_0 and always using the first edge on every adjacency list, until a back-edge is encountered, we will have computed the spine of the cycle $C(e_0)$ and that is the procedure followed in line (3). By following the tree-path from $w_0 = lowpt[y]$ to the vertex x, one can compute the stem of the cycle $C(e_0)$, if needed. We consider that the computed spine of $C(e_0)$ is comprised by vertices $\{y, \ldots, w_k\}$.

After computing the spine of $C(e_0)$, the algorithm discovers all edges that emanate from it. Since, the spine itself is constructed by taking the first edge that exists in the adjacency list of each visited vertex, all other edges that exist in the adjacency list of each vertex which is included in the spine are emanating from it. Therefore, all outgoing edges, except the first one, of each vertex in the spine of $C(e_0)$, are examined iteratively by the algorithm in the loop which starts at line (4). The vertices of the spine are visited in a bottom-up fashion, considering as bottom the leaves of the DFS sub-tree that corresponds to the segment, namely examining vertices w_k, w_{k-1}, w_{k-2} until the endpoint of e_0 is reached. For each of the edges that emanate from a vertex, the procedure check_strong_planarity is called. Thus, segments $S((w_k,))$ – i.e. the segments that emanate from w_k – are tested first, $S((w_{k-1},))$ are tested next and so on until segments $S((y,))$ are tested or conditions of non-(strongly)-planarity of the examined segments are found. The segments $S(e_i) = S((v,u_i))$ that emanate from the spine of $C(e_0)$ through a specific vertex v are examined in the order that the edges (v,u_i) are appeared in the adjacency list of v.

Each successful call to check_strong_planarity(e') returns the ordered list $A(e')$ of attachments of $S(e')$. Using this list according to section 2.3, we can check if segment $S(e')$ can be added to the interlacement graph $IG(C(e))$ and in which side of the cycle $C(e)$ without interlacements. In case that the examined segment $S(e')$ were not planar or not strongly planar, execution would have stopped inside the nested call of check_strong_planarity(e'). Here, only the interlacement of $S(e')$ with the already examined part of $S(e_0)$ is tested. Therefore, if control reaches line (14), all examined sub-segments are combined without interlacements in the computed interlacement graph $IG(C(e))$. What remains to be tested for $S(e_0)$ is whether it is strongly planar, i.e. whether its edges do not encircle the stem of $C(e_0)$, using the procedure again described in section 2.3. At the same time, using $A(e')$, we can update the ordered list L of attachments of the whole segment $S(e_0)$. Finally, L is returned.

So, one may finally note that ok, the above procedure checks a segment for strong planarity, but how do we test the whole graph? Well, since the input graph is bi-connected, there is exactly one edge out of vertex 1 (on the graph DFS graph D_G computed for the input graph G), namely the edge (1,2). In case there was another edge, then its descendent sub-graph would not communicate with the sub-graph of edge (1,2), which happens by default in DFS structures produced for undirected graphs. This though, would render vertex 1 (the root) a cut vertex, since its removal would disconnect D_G, creating two independent sub-graphs. This cannot happen in a bi-connected graph and thus there cannot be other edges out of vertex 1, except for (1,2). As a result, $G = S(1,2)$ and it has been proven that G is planar if and only if $S(1,2)$ is strongly planar. Therefore, we call the procedure that checks for strong planarity for the segment $S(1,2)$ and if the result is positive then the graph G is planar and the embedding returned by the procedure is the desired planar embedding of the graph.

2.5 Construction of the embedding

We remind that in order to construct the planar embedding of a graph we need to reorder the adjacency lists of the graph's vertices so as the topology that they represent (among a vertex and its neighbors) induces a planar drawing, a schema without crossings. The procedure check_strong_planarity has already tested that such a reordering exists for the input graph. At the same time, it can be used to label the edges of the input graph by "*L*" or "*R*". This labeling $a(e)$ indicates that edge e is placed (together with the segment $S(e)$ that it creates) on the Left or Right side of the cycle it emanates from. This information, which is an outcome of the interlacement graph computed for each segment in order to check for strong planarity of the segment, will be exploited in order to construct the embedding of the graph. And why do we need this information? Because it encodes the topology that the graph must have in order to be planar, i.e. it shows where the edges must be placed related to the cycle they emanate from in order to yield a planar embedding. And why a differentiation on the side (left or right) of placement of an edge related to the cycle it emanates from produces a different order in the edge's adjacency list? We already know for sure that a difference in topology is reflected in the adjacency lists, but to be more specific: If a segment is transferred and placed in the opposite side of the cycle it emanates from, then the order of appearance of edges in the adjacency lists of vertices that are included in the segment is inversed, given that we maintain the invariant of a clockwise or counterclockwise order of visiting a node's neighbors. Intuitively, if a segment is placed in the opposite side of a given cycle, it becomes the mirrored image of itself. It is therefore a fact that depending on the side of the cycle that a segment $S(e)$ must be placed and thus depending on the labeling $a(e)$, the ordering of edges in the adjacency lists of the vertices of $S(e)$ must be different (reversed).

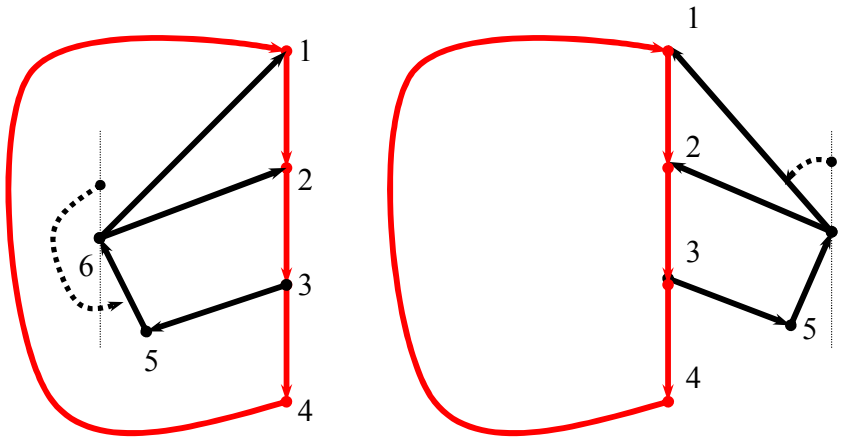


Figure 2.10: example of reversed canonical and canonical embedding

A strongly planar embedding of $S(e)$ will be called *"canonical"* (reversed canonical) if for all i, $0 < i < r$ such that w_i belongs to the stem of $C(e)$, the edge $\{w_i, w_{i+1}\}$ immediately follows (precedes) the edge $\{w_i, w_{i-1}\}$ in the counterclockwise ordering of edges incident to w_i, supposing that $e = (w_r, y)$ and $w_0, w_1, \ldots, w_r$ is the stem of $C(e)$. This implies that when the representation is canonical (reversed canonical) the line that is formed by the edges of the stem has one of its sides empty of segment's edges, i.e. all the edges of the segment are in the other side and that is why when we encounter edge $\{w_i,w_{i-1}\}$ ($\{w_i,w_{i+1}\}$) and continue visiting neighbors of w_i in the same manner / direction, the edge that follows and we discover next is

$\{w_i,w_{i+1}\}$ ($\{w_i,w_{i-1}\}$). This has a different meaning on the placement side of the segment $S(e)$ related to $C(e)$. If the edges (w_0,w_1), (w_1,w_2), …, (w_{r-1},w_r) are directed upwards then a (reversed) canonical representation means that the segment S(e) is placed in the (right) left side of the "line" that is formed by the above edges, while the opposite happens for a downward direction of the edges. In figure 2.10, the embedding of a segment $S(e = (3,5))$ is shown inside a given cycle $C(e_0 = (1,2))$. The cycle is comprised by edges (1,2), (2,3), (3,4), (4,1) while the segment $S(e)$ is comprised by edges (3,5), (5,6), (6,2), (6,1). Given the downward direction of the cycle's edges, the representation of the segment in the left side is reversed canonical, because when looking at node $w_i = 2$, which belongs to the stem of $C(e)$ we see that edge (w_i,w_{i+1}) immediately follows (w_i,w_{i-1}) when we check the adjacent edges of w_i counterclockwise. The opposite happens in the right schema of the figure, where the embedding is canonical. We can observe that the different embedding of the same segment causes the inversion of order of adjacent nodes in the adjacency lists of the segment's vertices. For example, vertex 6, when embedded in the left side of the cycle $C(e_0)$ has the neighbors {5,2,1} (in a counterclockwise order), while when embedded in the right side of the cycle $C(e_0)$ has an inversed neighborhood: {1,2,5}.

Considering all the above, to construct the embedding of a segment, we need to know in which side of the cycle it emanates from it must be placed and then, depending on the side, we construct a canonical or reversed canonical representation of the segment. The information about left or right placement of the segment comes from the check_strong_planarity procedure and the actual embedding is computed recursively, by further dividing the given segment into sub-segments and computing their embedding inside the higher levels' embedding, following the sane way we used to check. The procedure that implements the computation of the embedding is always called for a given edge e_0 and computes a canonical embedding of the segment $S(e_0)$ or a reversed canonical embedding, depending on the label $a(e)$.

To be more specific, to construct a canonical embedding of a segment $S(e)$ we draw the path w_0, …, w_k which consists the stem and spine of $C(e)$ as a vertical downwards path ("line") and we add the edge (w_k, w_0) to close the cycle. Then, for i, $1 \le i \le m$ and $a(e_i) = R$ we extend the embedding of $C + S(e_1) + \ldots + S(e_{i-1})$ by glueing a canonical embedding of $S(e_i)$ onto the right side of the vertical path, and for i, $1 \le i \le m$ and $a(e_i) = L$ we extend the embedding of $C + S(e_1) + \ldots + S(e_{i-1})$ by glueing a reversed canonical embedding of $S(e_i)$ onto the left side of the vertical path. Similarly, if the goal is to construct a reversed canonical embedding of $S(e_0)$, then if $a(e_i) = L$, then a canonical embedding of $S(e_i)$ is glued onto the right side of the vertical path, and if $a(e_i) = R$, then a reversed canonical embedding of $S(e_i)$ is glued onto the left side of the vertical path.

References

[AP61] L. Auslander and S. V. Parter. On embedding graphs in the sphere. Journal of Mathematics and Mechanics, 10(3):517-523, 1961.

[Bak94] B. Baker. Approximation algorithms for NP-complete problems on planar graphs. Journal of the Association for Computing Machinery, 41(1):153-180, 1994.

[BL76] K. Booth and G. Lueker. Testing for the consecutive ones property, interval graphs and graph planarity using PQ-tree algorithms. Journal of Computing and System Sciences, 13:335-379, 1976.

[BM90] D. Bienstock and C. Monma. On the complexity of embedding planar graphs to minimize certain distance measures. Algorithmica, 5(1):93-109, 1990.

[BM99] J. Boyer and W. Myrvold. Stop minding your P's and Q's: A simplified O(n) planar embedding algorithm. In SIAM-ACM Symposium on Discrete Algorithms, pages 140- 146, 1999.

[BT96] G. Di Battista and R. Tamassia. On-line planarity testing. SIAM J. Computing, 25(5), 1996.

[CNAO85] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using pq-trees. J. Comput. System Sci., 30:54-76, 1985.

[DMP64] G. Demoucron, Y. Malgrange, and R. Pertuiset. Graphes planaires: reconnaissance et construction de representations planaires topologiques. Rev. Francaise Recherche Operationnelle, 8:33-47, 1964.

[ET76] S. Even and R. E. Tarjan. Computing an st-numbering. Theoretical Computer Science, 2:436-441, 1976.

[HT74] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. J. Association Computing Machinery, 21(4):549-568, 1974.

[LEC67] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In Theory of Graphs, International Symposium Rome 1966, pages 215-232. Gordon and Breach, 1967.

[LT79] R. Lipton and R. Tarjan. A separator theorem for planar graphs. SIAM J. Appl. Math., 36(2):177-189, 1979.

[LT80] R. Lipton and R. Tarjan. Applications of a planar separator theorem. SIAM J. Comput., 9(3):615-627, 1980.

[MM96] K. Mehlhorn and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. Algorithmica, 16:233-242, 1996.

[SH92] W.-K. Shih and W.-L. Hsu. A simple test for planar graphs. In Proceedings of the Sixth Workshop on Discrete Mathematics and Theory of Computation, pages 35-42, 1992.