Chapter 5
Connectivity in graphs

Introduction

This chapter references to graph connectivity and the algorithms used to distinguish that connectivity. Graph connectivity theory are essential in network applications, routing transportation networks, network tolerance e.t.c. Separation edges and vertices correspond to single points of failure in a network, and hence we often wish to identify them. We are going to study mostly 2-connected and rarely 3-connected graphs.

5.1 Basic Definitions

- A connected graph is an undirected graph that has a path between every pair of vertices

- A connected graph with at least 3 vertices is 1-connected if the removal of 1 vertex disconnects the graph

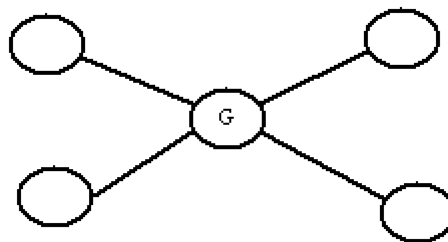


Figure 5.1.The removal of g disconnects the graph.

- Similarly, a graph is one edge connected if the removal of one edge disconnects the graph.

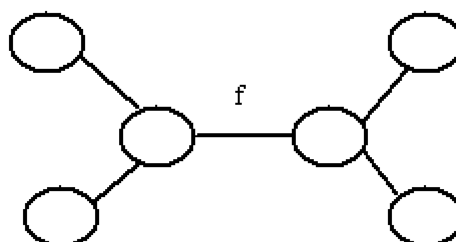


Figure 5.2.The removal of edge f disconnects the graph

5.2 Vertex Connectivity Vs. edge connectivity

Connectivity based on edges gives a more stable form of a graph than a vertex based one. This happens because each vertex of a connected graph can be attached to one or more edges. The removal of that vertex has the same effect with the removal of all these attached edges. As a result, a graph that is one edge connected it is one vertex connected too.

For example

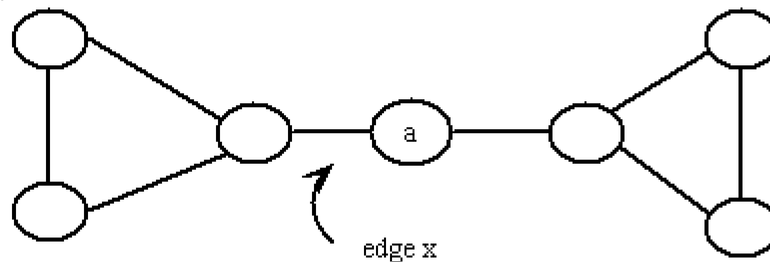


Figure 5.3.The removal of vertex *a* f disconnects the graph

As shown in Figure 5.3 graph *g* is one edge and one vertex connected. The removal of vertex *x* has the same effect with a possible removal of vertex *a* (with the term "effect" we mean the graph disconnection). The same definitions apply to *k*-connected graphs

1. A connected graph is *k*-connected if the removal of *k* vertices disconnects the graph.
2. A *k*-edges connected graph is disconnected by removing *k* edges

Note that if *g* is a connected graph we call *separation edge* of *g* an edge whose removal disconnects *g* and *separation vertex* a vertex whose removal disconnects *g*.

5.3 Bi-connectivity

5.3.1 Bi-connected graphs

Lemma 5.1: Specification of a *k*-connected graph is a bi-connected graph (2-connected).A connected graph *g* is bi-connected if for any two vertices *u* and *v* of g there are two disjoint paths between *u* and *v*. That is two paths sharing no common edges or vertices except *u* and *v*.

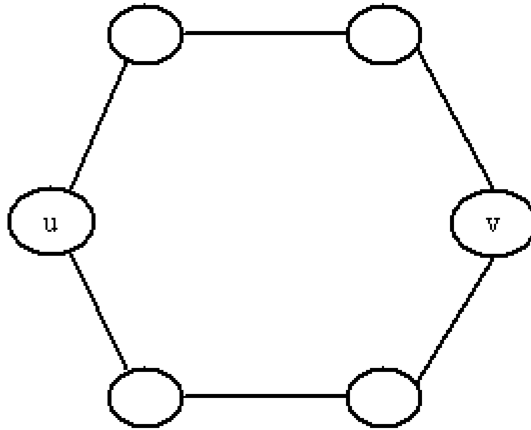


Figure 5.4.a bi-connected graph

Theorem5.1. For any two vertices of a bi-connected graph *g* there is a simple cycle containing them

Proof. Let's assume that there is no cycle. Then there is only one path from *u* to *v*. If we remove that path we disconnect the graph. That means that the graph is one-connected. We have a contradiction because we supposed that we have 2-connected graph.

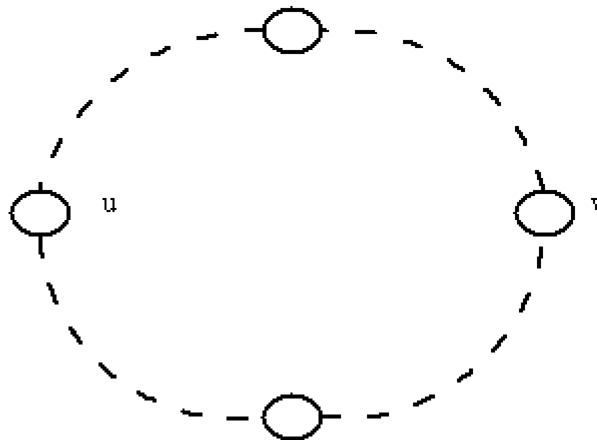


Figure 5.5: One 2-connected graph

5.3.2 Bi-connected components

The study of bi-connected components is important in computer networks where edges represent connection. Even if a router in a bi-connected component fails, messages can still be routed in that component using the remaining routers.

A bi-connected component of a graph *g* is a sub-graph satisfying one of the following:

1. It is a maximal sub-graph of g that is bi-connected (Maximal: If we add any other vertex or edge the graph does not remain bi-connected)
2. A single edge of g consisting of a separation edge and its end-points

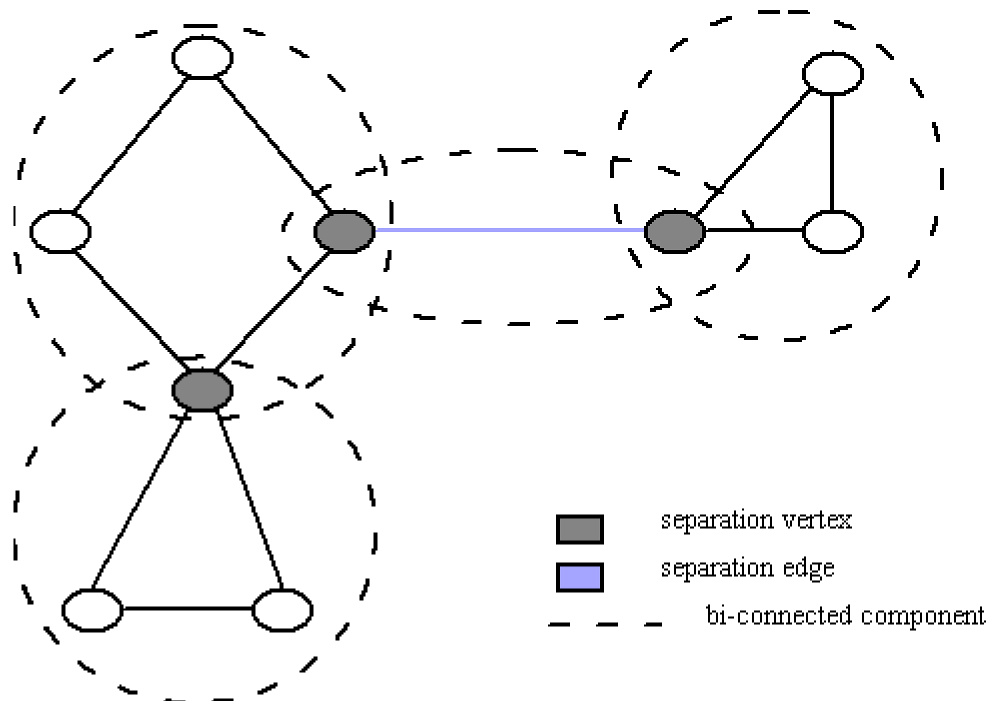


Figure 5.6. Bi-connected components, bridges and articulation points

Separation edges are also called *bridges* and separation vertices are also called *articulation points* As shown in Figure 5.6 let g be a graph. An articulation point is a vertex whose removal disconnects the graph and a bridge is an edge whose removal disconnects the graph

Let $G=(V, E)$ be a depth-first tree of G as shown in Figure 5.6. The articulation points are the heavily shaded vertices, the bridges are the heavily shaded edges and the bi-connected components are the edges in the cycled regions with the numbering shown.

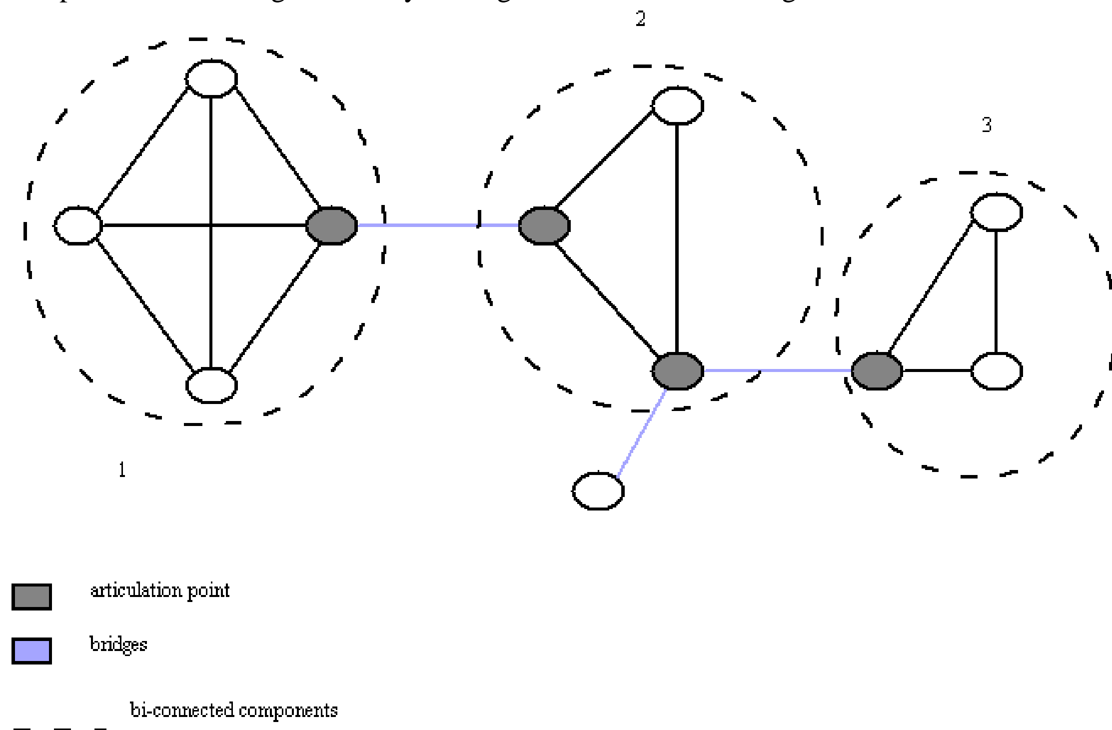


Figure 5.6.the articulation points, bridges and bi-connected components of a connected, undirected graph.

We can determine articulation points, bridges and bi-connected components using the depth-first-search algorithm

Theorem5.2 The root of the DFS is an articulation point if and only if it has two or more children.

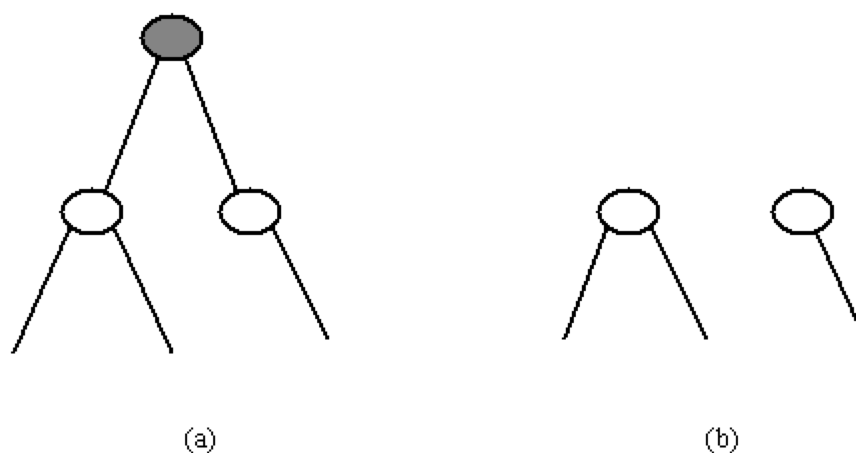


Figure 5.7. (a) The DFS tree (b) The tree after the removal of the heavily shaded root

Proof. Since there are no cross edges between the subtrees of the root if the root has two or more children then it is articulation point since its removal separates these two subtrees. If the root has only one child then its removal does not disconnect the DFS tree and as a result cannot disconnect the graph in general.

Theorem5.3 Let *u* be a non-root vertex in *Gp*. Then *?* is an articulation point of *G* if and only if there is no back edge *(u, w)* such that in *Gp* , *u* is a descendant of *?* and *w* is a proper ancestor of *?*.

Proof: Let consider the typical case of a vertex *u*, where *u* is not a leaf and *u* is not the root. Let *?*1, *?*2,...., *??* be the children of *u*. For each child there is a sub-tree of the DFS tree rooted at this child. If for some child, there is no back edge (because *G* is undirected we cannot distinguish between back edges and forward edges, and we all call them back edges) going to a proper ancestor of *u* , then if we are to remove i , the sub-tree would become disconnected from the rest of the graph, and hence *u* is an articulation point. On the other hand if every one of the sub-trees rooted at the children of *u*, have back edges to proper ancestors of *u* , then if *u* is removed the graph remains connected

The leaves cannot be articulation points because if we remove one leaf the rest of the tree remains connected.

5.4 The algorithm for identifying articulation points

All previous theorems and lemmas provide us with the proper background to identify articulation points. We can design an algorithm to check these conditions.

The first thing we have to check is if there is a back edge from a sub-tree to an ancestor of a given vertex. It would be too expensive to keep track of all the back edges from each sub-tree because may be *?*(*e*) back edges. A more simple solution is to keep track of back edge that goes highest in the tree.(closest to the root). As we travel from *u* towards the root the discovery times of these ancestors of *u* get smaller and smaller. So we keep track of the back edge (*?, w*) that has the smallest value of $d[w]$.

We define $low[u] = \min \begin{cases} d[u] \\ d[w] : (u, v) \end{cases}$

Where (*u, w*) is a back edge for some descendant *u* of *?*

$low[u]$ is the highest (closest to the root) that you can get in the tree by taking any one back edge from either u, or any of its descendants.

Initialization:

$low[u] = d[u]$

Back edge (*u, ?*)*:*

$low[u] = \min(\ low[u], d[u])$. We have detected a new back edge coming out of u. If this goes to a lower d value than the previous back edge then make this the new low.

Tree edge (*u, ?*)*:*

$low[u] = \min(\ low[u], low[u])$ .Since *?* is in the sub-tree rooted at u any single back edge leaving the tree rooted at *?* is a single back edge for the tree rooted at *u*.

Once *low*[*u*] is computed for all vertic es *u*, we can test whether a given non-root vertex *u* is an articulation point by using these simple steps. *u* is an articulation point if and only if has a child in the DFS tree for which $low[u] \geq d[u]$ since if there were a back edge from either *?* or one of its descendants to an ancestor of *?* then we would have $low[u] < d[u]$

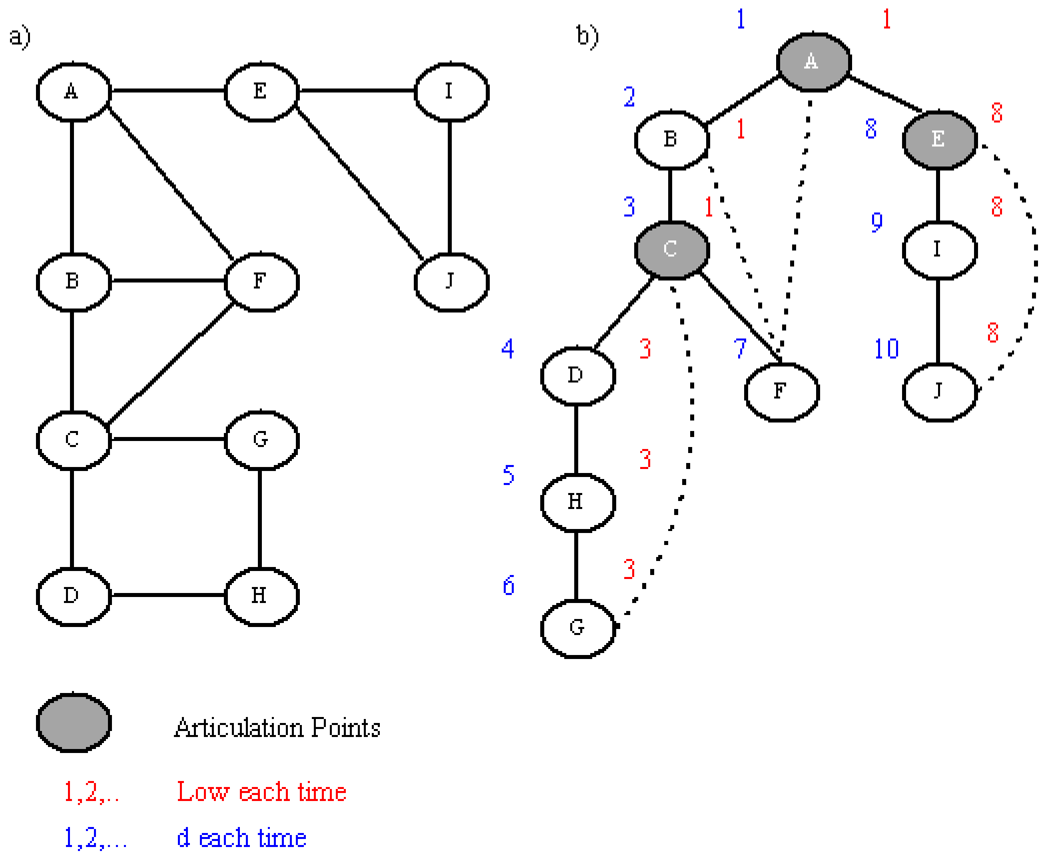


Figure 5.8. The DFS tree of (a)

The complete algorithm for computing articulation points is

Articulation points
Input: DFS tree
Output: The set of artic ulation points

Algorithm ArtPt (u)

1. $color[u] \leftarrow gray$
2. $low[u] \leftarrow time{+}{+}$
3. $d[u] \leftarrow low[u]$
4. **for each** v **in** $Adj[u]$ **do**
5.   **if** $color[v] = white$ **then**   // (u, v) is a tree edge
6. $pred[v] = u$
7. $ArtPt(v)$
8. $low[u] = \min(low[u], low[v])$ //update $low[u]$
9.   **if** $pred[u] = null$ **then**   //root
10.   **if** this is u's second child **then**
11.   Add u to articulation points set
12.   **else if** $low[v] \geq d[u]$ **then**   //internal node

13 Add *u* to set of articulation points
14 **else if** $v \neq pred[u]$ **then** // (*u, v*) back edge
15 $low[u] = \min(low[u], d[v])$
16
17

When processing a vertex *u* we need to know when a given edge *(u, ?)* is a back-edge. To do this we check if *?* is colored gray. This is not quite correct because *?* may be the parent of *?* in the DFS tree and we are seeing the "other side" of the tree edge between *?* and *u*. So we must use the predecessor pointer to check that *?* is not the parent of *u* in the DFS tree.

In Figure 5.8 ,(b) is the DFS tree of the graph (a).Using the previous algorithm we can identify the articulation points. We start from vertex *a* where *low=1* and *d=1* and we color it gray. We follow the algorithm for the rest vertices and their *low* and *d* is shown in the figure. If we know for each vertex low and d we can easily find articulation points because *u* is an articulation point if and only if has a child in the DFS tree for which $low[u] \geq d[u]$.

As with DFS-based algorithms the running time is *?(n+e)*. You could use the algorithm to determine which edges are in the bi-connected components if we store the edges in a stack as we go through DFS search. When we come to an articulation point all the edges in the bi-connected component will be in the stack

5.5 Equivalence Classes and the Linked Relation

Let *C* be a collection of objects. We can define a Boolean relation for each pair *x*, *y* in *C.* The relation R is an equivalence relation if it has the following properties

I. *Reflexive*: *xRx* This means that the relation is true for each x in C
II. *Symmetric*: *xRy=yRx* for each pair *(x, y)* in C
III. *Transitive: xRy* and *yRz => xRz* . if *xRy* is true and *yRz* is true then *xRz* is true for every *x,y,z* in C

Two edges of a graph are linked if there is a cycle that contains them .A link relation is a sort of an equivalence relation.

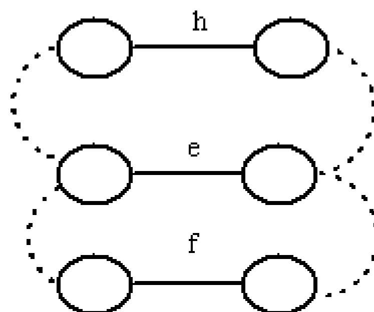


Figure 5.9. Transitive Property of Link Relation

In Figure 5.9, it is obvious that if there is a cycle that contains *h* and *e* and another that contains *e* and *f*. The two cycles have a common edge and if we remove *e* we will still have a

cycle that contains *h* and *f* hence the link relation is transitive. It is also obvious that the link relation is reflexive and symmetric so link relation is an equivalence relation.

5.6 Bi-connected components computing via DFS

In the beginning we can construct an auxiliary graph *B* as follows

An Auxiliary graph B of a given graph G has the following properties:

- The vertices of *B* are the edges of *G*.
- For every back-edge *e* of G let *f1,f2,...,fk* be the discovery edges of *G* that form a cycle with e. Graph B contains the edges *(e,f1),...,(e,fk)*

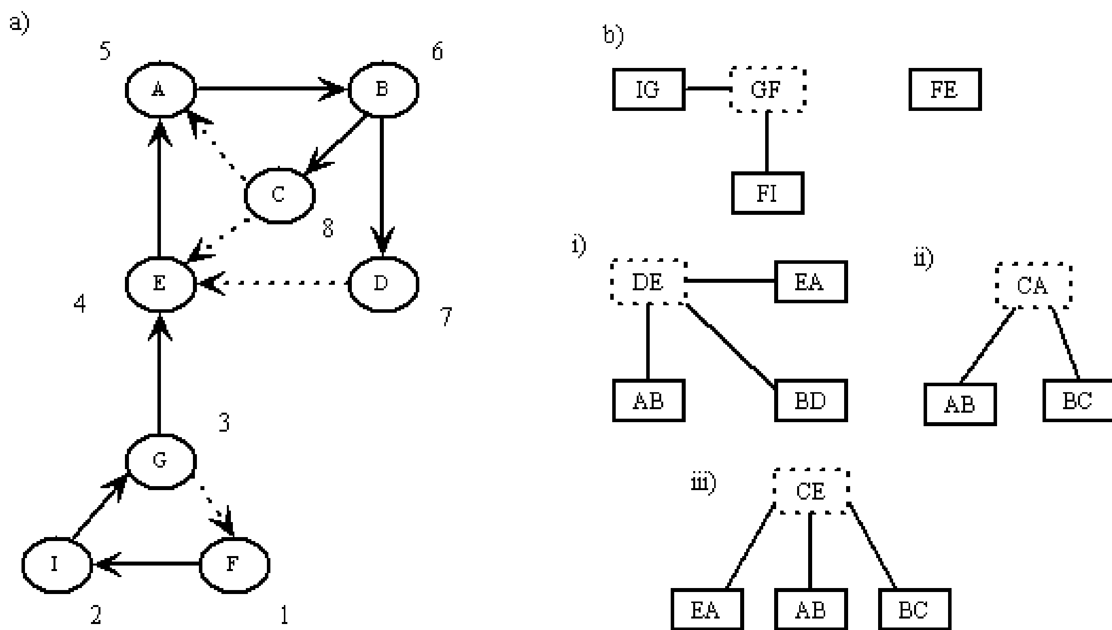


Figure 5.10. The steps of construction of the auxiliary graph from the given graph a

In Figure 5.10 (a) is the given graph. In (b) vertices are becoming edges and for every back edge we include in the Auxiliary graph the edges that form a cycle with these back edges. For example we have the back edge *GF* which forms a cycle with edges *FI* and *IG*. As a result, in the Auxiliary graph we have (i).If we combine (iii), (iv) and (v), we get the final auxiliary graph shown in figure 10.

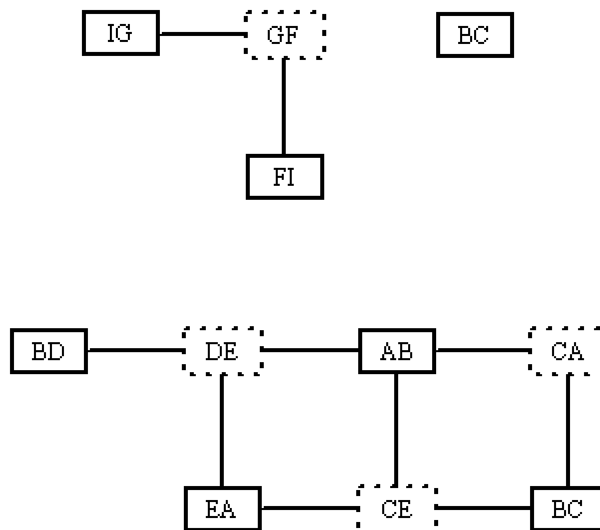


Figure 5.11. The final auxiliary graph

Since there are m-n+1 back edges and each cycle induced by a back-edge has at most $O(n)$ edges the graph has at most $O(nm)$ edges.

We can see that as a result of the previous step we have a "forest". Each connected component of this "forest" satisfies the equivalence class in the link relation. We can call these equivalence classes as link components of G.

We can summarize the previous steps in the following algorithm.

1. Perform a DFS on G.
2. Compute the auxiliary graph F by identifying the cycles of G induced by each back-edge.
3. Compute the connected components of F.
4. For each connected component of F output the vertices of G in the corresponding block.

The initial DFS traversal of G takes $O(m)$ time. The main computation however is the construction of the auxiliary graph. As a result algorithm takes $O(nm)$ time because the bottleneck is the computation of the auxiliary graph.

Now we have a simple way to determine the bi-connected components, separation edges and separation vertices of a graph G in linear time.

- The *bi-connected components* are the linked components of the auxiliary graph.

- The *separation edges* are the single – element link components of the auxiliary graph.

- A vertex v of G is a *separation vertex* if and only if v has incident edges in at least two distinct equivalence classes of linked edges

Note that we can simplify the algorithm in order to take $O(m)$ time using, a very important observation: we don't actually need the entire auxiliary graph but we only need to identify the connected components in B. As a result we don't actually need all the edges of the auxiliary graph but just enough of them in order to construct a spanning forest of B. So we can reduce the time in $O(m)$ by using a "smaller Auxiliary graph" which is a spanning forest of B.

Link Components
Input: A connected graph G
Output: The link components of G

Algorithm LinkComponents (G)

```
1       AuxGr ← null                          //Initially empty auxiliary graph.
2       DFS(s)                                //DFS of G starting at an arbitrary vertex s.
3       for each DFS discovery edge f
4               AuxGr ← AuxGr + f             //add f as vertex in Auxiliary graph
5               f ← unlinked                  //mark f unlinked
6       for each vertex v of G
7               p(v) ← parent(v)              //the parent of v in the DFS tree.
8               for each vertex v, in increasing rank order as visited in the DFS do
9                       for each back-edge e = (u, v) with destination v do
10                              AuxGr ← AuxGr + e
11                              while u ≠ s do
12                                      if f ∋ AuxGr corresponding to discovery edge (u, p (u)).
13                                              AuxGr ← AuxGr + (e, f)
14                                      if f = unlinked then
15                                              f ← linked
16                                              u ← p(u)
17                                      else
18                                              u ← s  //shortcut to the end of the while loop  19
        Compute the connected components of the Ayx
```

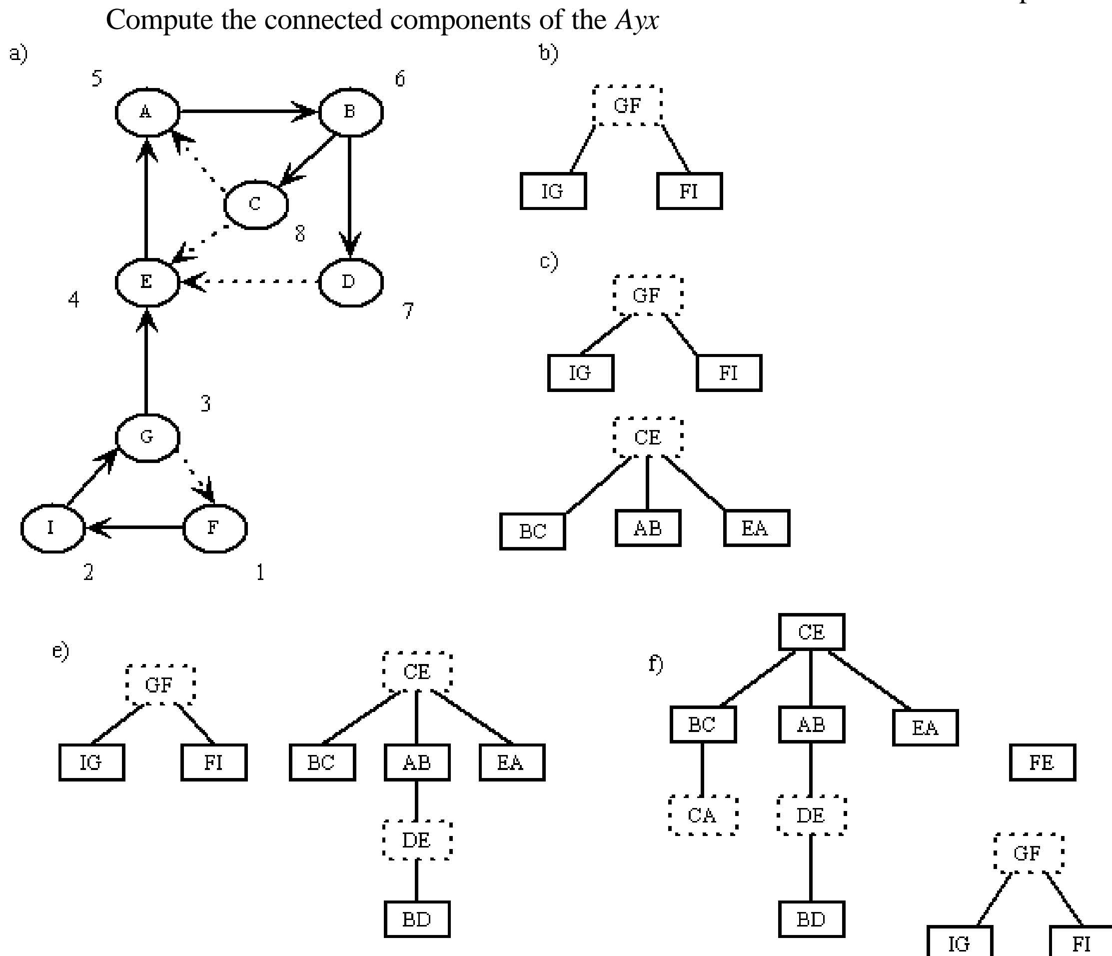


Figure 5.12. The algorithm in action

As we can see in Figure 5.12 (a) is the input graph *G* after the DFS traversal. Vertices are labeled by their rank in the visit order, and the back edges are drawn with dashed lines. In the beginning each discovery edge is inserted in the Auxiliary graph and marked as unlinked. After processing back edge (*G, F*) we have (b) and then after processing (*C, E*) we get (c) etc. The final Auxiliary graph is (f) in the end of the algorithm.

5.7 Fundamental circuits of a graph

- A *co-tree* of a graph $G = (V, E)$ with respect to a spanning tree $T = (V, E')$ is the set of edges $(E - E')$. If G has n vertices then any co-tree, if one exists, has $|E| - (n - 1)$ edges. Any edge of a co-tree is called a *chord* of the spanning tree.

- The *ring-sum* of two graphs *G1= (V1, E1)* and *G2= (V2,E2)* is the graph (*(V1UV2), ((E1 U E2) – (E1 n E2)))*. In other words the edge set of the ring-sum of *G1, G2* consists of those edges which are either in *G1* or are in *G2* but are not in both. The ring-sum of *G1* and *G2* is written like $G1 \oplus G2$

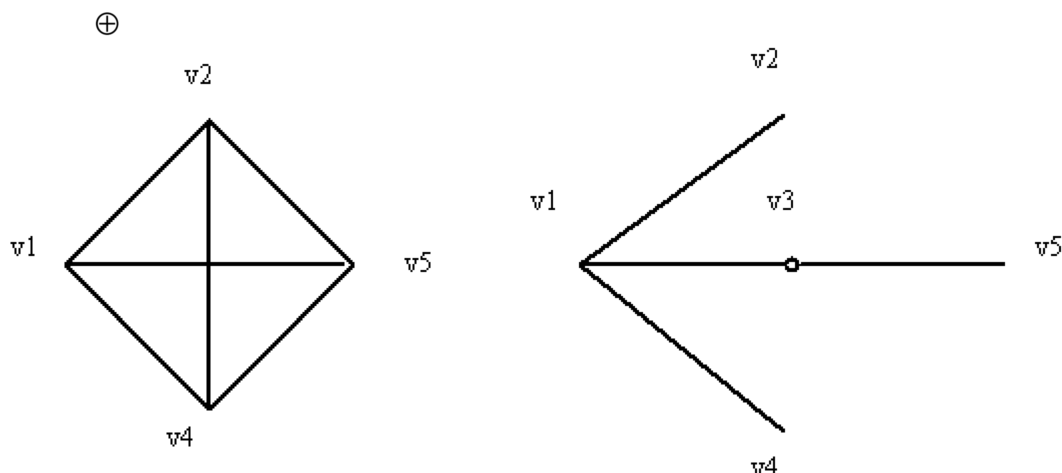
  It is easy to prove that the operation of ring-sum is both commutative and associative:

  $$G1 \oplus G2 = G2 \oplus G1,$$

  and

  $$(G1 \oplus G2) \oplus G3 = G1 \oplus (G2 \oplus G3)$$

It is easy to realize that the addition of a chord to a spanning tree of a graph creates precisely one circuit. In a graph the collection of these circuits with respect to a particular spanning tree is called asset of *fundamental circuits*. Any arbitrary circuit of the graph can be expressed as a ring-sum combination of the fundamental circuits. Hence we can suppose that the fundamental circuits form a *basis* for the circuit space.



(a) A given graph G

(b)A spanning tree of G

Figure 5.13. A spanning tree of a graph G

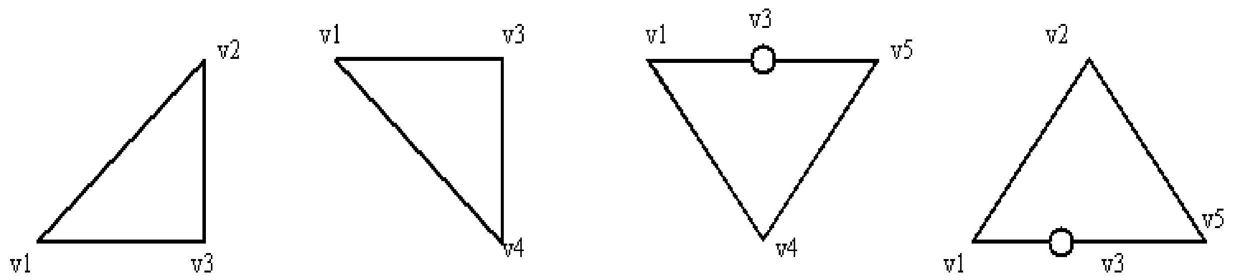


Figure 5.14. The fundamental set of circuits of *G* with respect to *T*

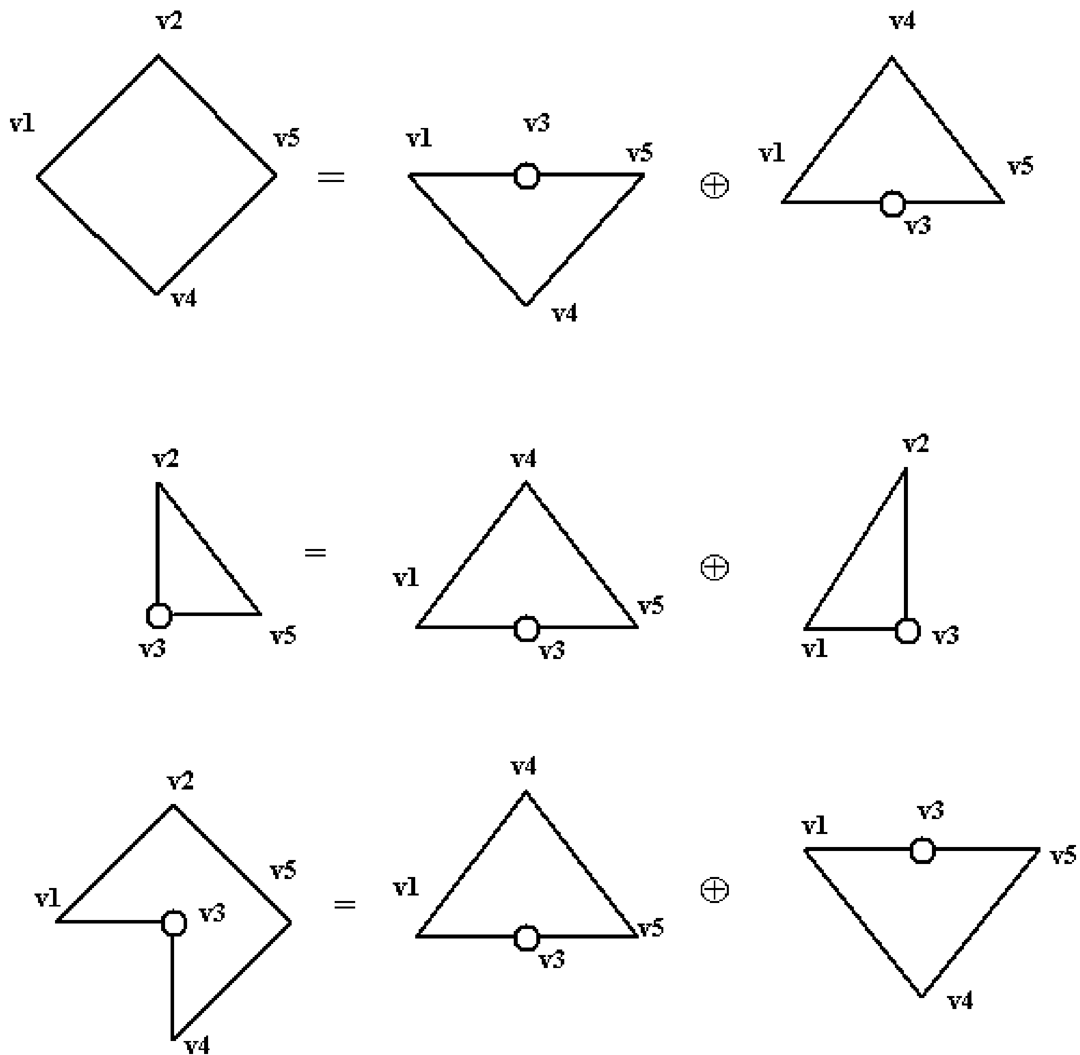


Figure 5.15. Circuits produced using right sum

All previous figures show for the graph illustrated ,a spanning tree T, the corresponding set of fundamental circuits and some other circuits expressed as linear combination of these. In general we have the following theorem

Theorem 5.4. A set of fundamental circuits, with respect to some spanning tree of a graph G, forms a basis for the circuit space of G.

We have an immediate corollary:

Corollary 5.1. The circuit space for a graph with $|E|$ edges and n vertices has dimension $(|E| - n + 1)$

We could construct an algorithm that can find a set of fundamental circuits of a graph in polynomial time.

Set of Fundamental circuits
Input: A graph G
Output: The set of fundamental circuits

Algorithm Fund (G)

1
2 Find a spanning tree and the corresponding co-tree CT of G
3 $FCS \leftarrow null$
4 **for all** $e_i = (u_i, u_i') \in C$ **do**
5 find the path from u_i to u_i' in T and denote it by P_i
6 $C_i \leftarrow P_i \cup \{e_i\}$
7 $FCS \leftarrow FCS \cup C_i$
8

Firstly we find the T spanning tree and the corresponding co-tree CT of G. We can achieve that in $O(\max(n, |E|))$ time. Then for each edge in CT the algorithm finds one fundamental circuit and adds in to the set FCS. The whole path finding process requires no more than $O(n)$ time. Because the number of the edges in CT is $O(|E|)$,the total worst complexity of the algorithm is $O(n^3)$.

Fundamental circuits were early used by Kirchoff to develop its voltage laws. These laws are based in solving several equations over an electric network. Theorem 5.4 and its corollary tell us which circuits of the underlying graph of the network and how many of them, provide a linearly independent set of equations.