```
 ------------        -----
| root        |      /     |
|   o         |     /      |
|    \        |    /       |
|      .      |   /        |
|       .     |  /         |
|  B1     .   | /    B3    |
|             |/           |
 ------------o------------
         v  /|\
           / | \
          /  |w \
         /   o   \
        /         \
       /           \
      /             \
     /       B2      \
    /                 \
   /                   \
  /                     \
  -----------------------
```

Figure 5-19. Block Diagram.

Figure 5-16. Trace of Strong Components Algorithm from v_1.

```
               v o<----------
                 |           |
                 |           |
          P(v,u) |           |
                 |           |
                 |           |
               u o           |
                /|\          |
               / | \         |
              /  |  \        |
        ---------    \       |
       |    /         \    z o
Q(u,v) |   /           \     ^
       |  /          w  \    |
        ------------->o------
        /                 \
       /    T(u)           \
      /                     \
```

```
  T(u):  Search subtree rooted at u.

P(v,u):  Search path from v to u.

Q(u,v):  Path from u to v.

     w:  Last vertex in T(u) ï Q(u,v).

     z:  Successor of w on Q(u,v).
```

Figure 5-17. Diagram for Correctness Argument.

```
C1 o<------o C2
          ^|
         / |
        /  |
       /   |
      /    |
     /     |
    /      v
C4 o------>o C3
```

Figure 5-18. Condensation Digraph for G.

Section 5-5

v_1,v_2,v_3	v_1,v_2,v_3	Scan (v_3,v_2)		v_3:2
v_1,v_2,v_3	v_1,v_2,v_3	Pop v_3		
v_1,v_2	v_1,v_2,v_3	Scan (v_2,v_4) Push v_4	v_4:4	v_4:4
v_1,v_2,v_4	v_1,v_2,v_3,v_4	Display $\{v_4\}$ and Pop v_4 from path. Pop list up to v_4.		
v_1,v_2	v_1,v_2,v_3	Scan (v_2,v_5) Push v_5	v_5:5	v_5:5
v_1,v_2,v_5	v_1,v_2,v_3,v_5	Scan (v_5,v_1) Scan		v_5:1
(v_5,v_6)	v_6:6 v_6:6	Push v_6		
v_1,v_2,v_5,v_6	v_1,v_2,v_3,v_5,v_6	Scan (v_6,v_7) Push v_7	v_7:7	v_7:7
v_1,v_2,v_5,v_6,v_7	v_1,v_2,v_3,v_5,v_6,v_7	Scan (v_7,v_8) Push v_8	v_8:8	v_8:8
v_1,v_2,v_5,v_6,v_7,v_8	$v_1,v_2,v_3,v_5,v_6,v_7,v_8$	Scan (v_8,v_6)		v_8:6
v_1,v_2,v_5,v_6,v_7,v_8	$v_1,v_2,v_3,v_5,v_6,v_7,v_8$	Pop v_8		v_7:6
v_1,v_2,v_5,v_6,v_7	$v_1,v_2,v_3,v_5,v_6,v_7,v_8$	Pop v_7		
v_1,v_2,v_5,v_6	$v_1,v_2,v_3,v_5,v_6,v_7,v_8$	Pop v_6 from path. Display and Pop $\{v_6,v_7,v_8\}$ from list.		
v_1,v_2,v_5	v_1,v_2,v_3,v_5	Pop v_5		v_2:1
v_1,v_2		Pop v_2		
v_1	v_1,v_2,v_3,v_5	Pop v_1 from path. Display and Pop$\{v_1,v_2,v_3,v_5\}$ from list.		
Empty	Empty	Exit.		

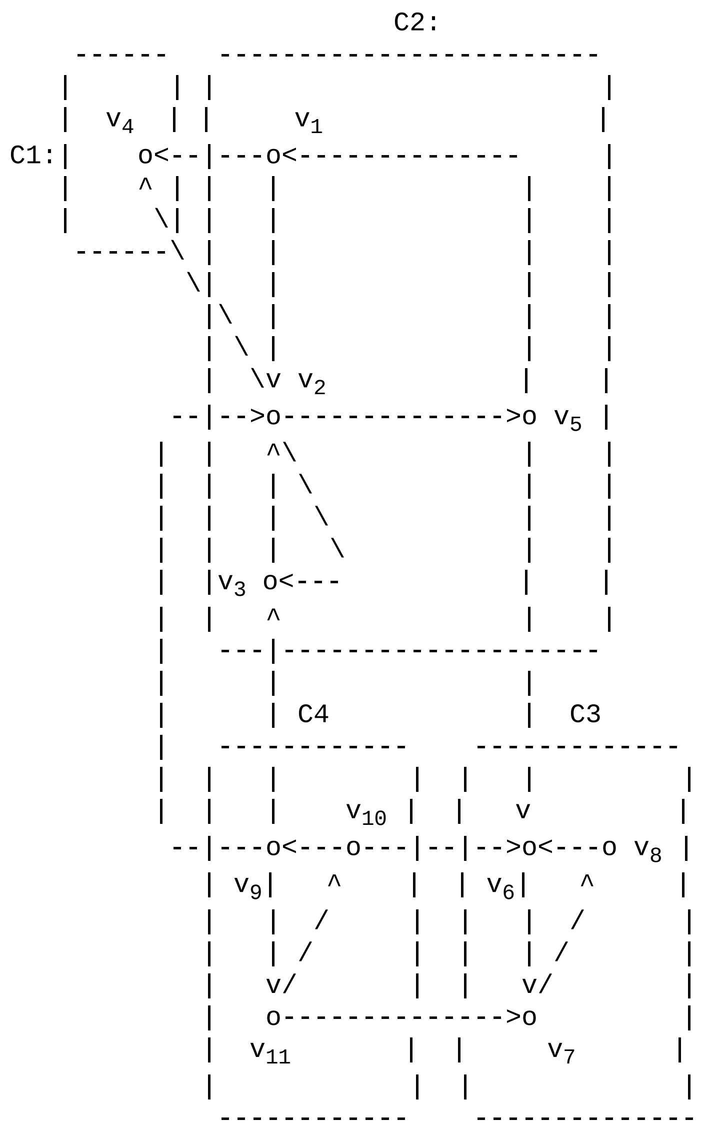
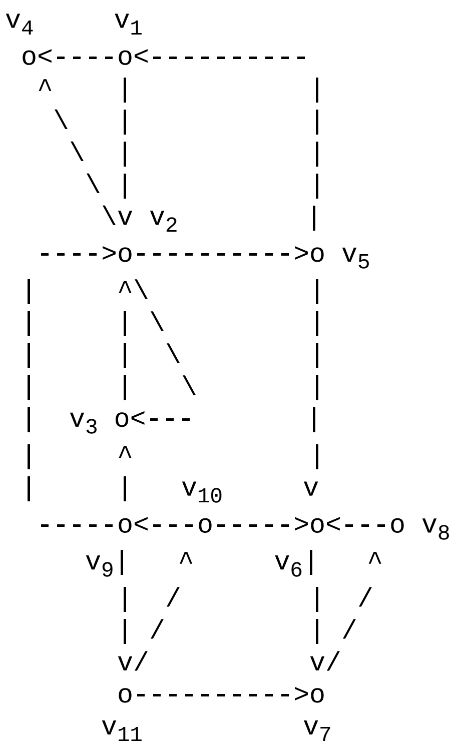


Figure 5-15. Strong Components of G.

SEARCH PATH	COMPONENT LIST	ACTION	Dfsnum	Low
v_1	v_1	Initialization	v_1:1	v_1:1
v_1	v_1	Scan (v_1,v_2) Push v_2	v_2:2	v_2:2
v_1,v_2	v_1,v_2	Scan (v_2,v_3) Push v_3	v_3:3	v_3:3

SECTION 5-4



Adjacency List:

v_1:	v_2, v_4	v_7:	v_8
v_2:	v_3,v_4,v_5	v_8:	v_6
v_3:	v_2	v_9:	v_2,v_3,v_{11}
v_4:	nil	v_{10}:	v_6,v_9
v_5:	v_1,v_6	v_{11}:	v_7,v_{10}
v_6:	v_7		

Figure 5-14. Digraph G.

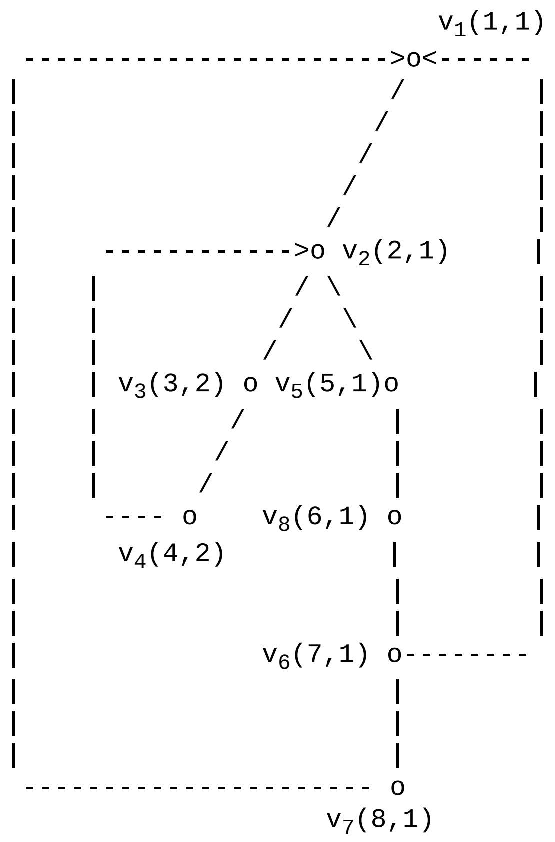
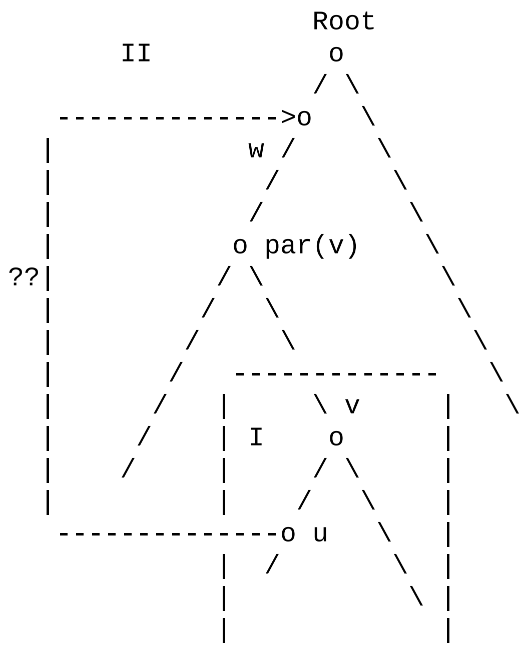


Figure 5-13. Graph with (Dfsnum,Low) Values at Each Vertex.



I: The subtree rooted at v.

II: The remainder of the search tree.

Figure 5-11. Illustration for Trapping Condition.

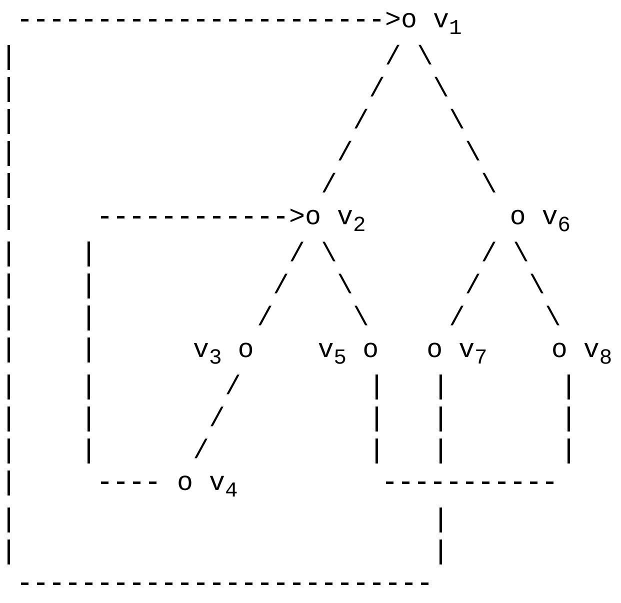


Figure 5-12. Graph for Orientation Example.

```
 ----------------------->o 1,1
|                       / \
|                      /   \
|                     /     \
|                    /       \
|                   /         \
|     ------------>o 2,2       o 6,1
|    |            / \         / \
|    |           /   \       /   \
|    |          /     \     /     \
|    |     3,2 o   5,5 o   o 7,1   o 8,8
|    |        /            |
|    |       /             |
|    |      /              |
|     ---- o 4,2           |
|                          |
|                          |
 --------------------------
```

Figure 5-9. Labelling with (Dfsnum, Low).

```
              w
Ancestor of v o<--------------
             / \              |
            /   \             |
           /     \Par(v)      |
          /       o           | B
         /         \ T        |
        /           \         |
       /             \        |
      /               \  v    |
            ---------->o------
           |            \
           |             \
           |            T \
         B |               o u - Child of v
           |                \
           |                 \
           |                  \
            -------------------o  z - Descendant of v
```

Figure 5-10. Relations among Ancestors, Descendants, and Children of v.

```
                      |
                      v
       ---------------o<-------------
      |                              |
      |                              |
      |                              |
      v                              |
Get Next Edge x -------------> G - x Connected?
      |                              |
      |                              |
      | No More                      | No
      |                              |
      v                              v
  Orientable                   Not Orientable
      |                              |
      |                              |
      |                              |
       -------------->o<-------------
                      |
                      v
```

Figure 5-7. Exhaustive Search Algorithm for Orientability.

```
    Root
     o
    / \
   /   \
  /   w o<----------------
 /       \                |
/         \               |
           \              | BE(u,w)
          v o             |
           / \            |
          /   \ TP(v,u)   |
         /     \          |
        /     u o---------
       /         \
```

TP(v,u): Tree path from v to u

BE(u,w): Back edge from u to w

Figure 5-8. Induced Path from v to Some Ancestor w of v.

F: Forward, B: Back, C: Cross, Unlabelled: Tree

(b) Directed Edge Classification for Graph of (a).

Figure 5-5. Directed Edge Classification.

LM7¯®RM73¯®JU¯SECTION 5-3

```
o              o
|\            /|
| \          / |
|  \        /  |
|   \      /   |
|    o----o    |
|   /      \   |
|  /        \  |
| /          \ |
|/            \|
o              o
```

(a) Non-orientable Graph.

```
o         o         o         o
|\       /|         ^\       /^
| \     / |         | \     / |
|  \   /  |         |  \   /  |
|   \ /   |         |   v v   |
|    o    |         |    o    |
|   / \   |         |   / \   |
|  /   \  |         |  /   \  |
| /     \ |         | /     \ |
|/       \|         |v       v|
o         o         o         o
```

Orientable Graph G   Orientation of G.

(b)

Figure 5-6.   Orientation.

Figure 5-4. Edge Classification for Directed Graphs.

```
           v8      v9
    ------->o----->o
   |        ^      |
   |        |      |
   |        |      |
v4 |     v2 |      v
   o------->o<-----o v10
   ^        ^     ^
   |        |    /
   |        |   /
   |        |  /
   |        | /
   o<-------o/<----o v7
 v3      v1 ^      ^
            |      |
            |      |
            |      |
            o<-----o
           v6       v5
```

(a) Directed Depth First Search Edge Classification.

```
                           -----------------------
                          /         C             |
                      v1 v                        |
   ---------------------o <---------------        |
  |                    / \          C     |       |
  |                   /   \               |       |
  |                  v     v              |       |
  |     -------> v2 o       o v3          |   o v5|
  |    |           /^        \            |  / \  |
  |    |          / |         \           | /   \ |
  |    |         v  |    C     v          |v     v|
  |    |     v8 o    ---------- o v4   v6 o       o v7
F |  B |       / ^              |
  |    |      /   \      C      |
  |    |     v     -------------
  |    | v9 o
  |    |   /
  |    |  /
  |    | /
  |    |v
   --> o
      v10
```

(c) Breadth First Search Tree for (a), Assuming

Adjacency Lists Numerically Ordered.

Figure 5-2. Depth and Breadth First Search Examples.

®

```
             -- Tree edge (1st visit)
            |
            |yes
            |
            |
    y     --                       -- Tree edge (2nd visit)
unexplored  |                      |
            |no                    |yes
            |                      |
            |  y the predecessor   |
             --of x on the depth --              --Back edge
               first search path?  |             |  (1st visit)
                                   |             |
                                   |no           |yes
                                   |             |
                                    --Dfsnum(x)--
                                      >Dfsnum(y) |
                                                 |no
                                                 |
                                                 |
                                                  --Back edge
                                                    (2nd visit)
```

Figure 5-3. Edge Classification Decision Tree for Edge (x,y).

```
                -- Tree edge
               |
               |Yes
               |
               |
y unexplored --               -- Forward edge
               |             |
               |no           |yes
               |             |
                --Dfsnum(y) > -                  --Back edge
                  Dfsnum(x)?  |                  |
                              |no                |yes
                              |                  |
                               --   y an       --
                                 ancestor of x?  |
                                                 |no
                                                 |
                                                  --Cross edge
```

(a) Example Graph G.

®

```
                        v1(1)
                          o
                         / \
                        /   \
                       v     v
                v2(2) o       o v5(8)
                     / \       \
                    /   \       \
                   /     \       \
                  v       v       v
                 o v4(3)   o v8(5) o v7(8)
                /         /         \
               /         /           \
              /         /             \
             v         v               v
            o         o v9(6)           o v6(9)
         v3(4)       /
                    /
                   v
                  o v10(7)
```

(b) Depth First Search Tree for Graph of (a)

Assuming Adjacency Lists Numerically Ordered.

```
                        v1
               ----------o----------       Level 0
              |         / \         |
              |        /   \        |
              |       /     \       |
              v      v       v      v
           v2 o   v3 o    v5 o  v6 o      Level 1
             /|\              |
            / | \             |
           /  |  \            |
          /   |   \           |
         /    |    \          |
        v     v     v        v
    v4 o   v8 o      o v10   o v7         Level 2
              |
              |
              |
              v
              o v9                        Level 3
```

```
                         |
                         |
                      Number u
                      Push (S,u)
                         |
                         v
                         o<-----------------------------
                         |                              |
                         v                              |
                         o<----------------------       |
                         |                       |      |
                         |       (Advance)       |      |
                         v                       |      |
                 Next(Top(S),v)--------------->Push(S,v)|
                         |          Succeed    Number v |
                         |Fail                          |
                         |                              |
                         |       (Backup)               |
                         v                              |
                  Empty(Pop(S))------------------------->
                         |           Fail
                         |
                         |Succeed
                         |
                         v
```

Figure 5-1. Depth First Search Using Explicit Stack.

```
                        v8      v9
                         o-------o
                         |       |
                         |       |
                         |       |
               v4     v2 |       |
                o--------o-------o v10
                |        |
                |        |
                |        |
                |        |
                |        |
                o--------o-------o v5
              v3      v1 |       |
                         |       |
                         |       |
                         |       |
                         o-------o
                        v6      v7
```

the performance of Dijkstra's algorithm on the same graphs, as a function of the sparseness (edge density) of the graphs.

(10) How do depth-first search and breadth-first search interface differently with the block structure of a graph?

(11) Carefully follow the logic of the strong components algorithm as it processes a directed tree rooted at v. Check whether each step is correctly implemented.

(12) Let R be the reachability matrix of a digraph G(V,E). Prove R®^{2}(i,i) equals the number of vertices in the strongly connected component containing vertex i.

(13) Suppose a digraph G(V,E) equals its transitive closure. Design an algorithm which determines whether G is strongly connected in O(|V|)time.

(14) Let G(V,E) be a graph. Design an algorithm that computes the minimum number of edges that have to be added to G to make it biconnected.

(15) Design an algorithm based on depth first search that computes the transitive closure of a digraph G(V,E) in time O(|V|®^{2} + |V||E|).

(16) Show how to use partitioning by blocks to more efficiently find shortest distances on a graph.

(17) How can partitioning using strong components be used to more efficiently find shortest distances on a digraph?

(18) Adapt the orientation algorithm to the case where the input graph is mixed, that is, some of the edges are directed and some are not.

(19) Prove the correctness of the block finding algorithm.

(20) Prove the following algorithm finds the strong components of a digraph G(V,E) (a) Repeat the following until every vertex of G has been labelled: perform a depth first search of G, but number the vertices in increasing order, as they are popped from the search stack; (b) reverse the directions on all the edges of G; repeat the same procedure as in step (a), restarting the search as necessary each time at the highest numbered vertex from step (a) not yet visited during this search. Prove that a pair of vertices are in the same strong component of G if and only if they are in the same depth first tree constructed in step (b).

of B2 at this point. The same pattern recurs when we scan B3. Eventually, after both B2 and B3 have been recognized, the search returns to complete the suspended traversal of B1. B1 is recognized when the search finally retracts from the search tree child of root.

Figure 5-19 here

CHAPTER 5: REFERENCES AND FURTHER READING

Depth-first search was introduced and applied to a variety of applications in Tarjan (1972) and (1974). Williamson (1984) uses depth-first search to extract Kuratowski subgraphs from nonplanar graphs in linear time. For a discussion of traversal techniques that avoid use of an explicit stack, such as Lindstrom's method, see Standish (1980). The alternative strong components algorithm in the exercises is from Basse (1988).

CHAPTER 5: EXERCISES

(1) Give an example of a class of graphs where the maximum stack length for depth-first search is O(|V|), while the maximum queue length for breadth-first search is O(1), and conversely.

(2) Perform depth-first search and breadth-first search on K(n), C(n), and K(m,n). What is the effect of the order of edges on the adjacency list on the traversal?

(3) Compare the performance for the recursive and the nonrecursive (stack) algorithm for depth first search on a set of test graphs; also compare the programming effort involved.

(4) If a depth first search starts at a vertex v, what is the maximum depth attained by the search stack in terms of some standard graphical invariants that depend on v? What is the minimax stack depth for the graph as a whole, over all possible starting vertices? What vertex would you start the search at in order to minimize the maximum stack depth?

(5) Assume that a depth first search tree has been constructed, and that then an edge of the graph is deleted. What happens if the search is repeated? What is the relation between the before and after search tree?

(6) Show that if a vertex u in an undirected G(V,E) is an ancestor (other than the parent) of v in the breadth-first search tree T on G, then (u,v) is not an edge of T. Contrast this with (undirected) depth first search, where if (u,v) is an edge, either u or v is an ancestor of the other.

(7) Adapt the breadth-first search algorithm to compute the level number of each vertex.

(8) Construct a graph where breadth first search would detect a cycle before depth-first search, and a graph where the reverse happens.

(9) Contrast the performance of the breadth first search algorithm for computing shortest paths on unweighted (that is, unit edge weight) graphs with

```
     Top, Pop: Stack Entry pointer function

Select some v in V(G)
Set Dfsnum(v) to Nextcount
Set Low(v)    to Dfsnum(v)
Create(SP);  Push (SP,v)
Create(BCL); Push (BCL,v)

repeat

   (* ADVANCE *)

   while   Next(Top(SP),u)  do

       if    Dfsnum(u) =  0

       then  (* Tree edge - Advance *)
             Push(SP,u); Push(BCL,u)
             Set Dfsnum(u) to Nextcount
             Set Low(u)    to Dfsnum (u)

       else  (* Back edge - Compare *)
             if   Dfsnum(u) < Dfsnum(Top(SP))
             then Set Low(Top(SP)) to min{Low(Top(SP)),Dfsnum(u)}

   (* BACK UP *)

   if    Top(SP) <> v

   then  Set    p = Predecessor_of_Top(SP)

         if     Low (Top(SP)) ò Dfsnum (p)

         then   (* Display block *)
                List and Pop BCL up to and including Top(SP)
                List p

         else   (* Inherit *)
                Set Low(p)  to  min {Low(p), Low(Top(SP))}

until   EMPTY (POP(SP))

End_Procedure_Block
```

For an example, refer to the diagram in Figure 5-19 which shows three blocks B1, B2, and B3 joined at an articulation point a. If the search starts at root, the block algorithm will eventually list the (vertex members of the) blocks B2 and B3, not necessarily in that order, and finally the members of block B1. The B2 block is recognized when the search retracts from w. In the search subtree T(v) rooted at v, w is the only child of v in B2. All the other descendants of v in B2 are descendants of w. By the time B2 is scanned, Low(w) has been set to Dfsnum(v). The algorithm uses this phenomenon to test for the completion of the traversal of a block, and lists the appropriate vertices components of G. A cut-vertex is also called an *articulation point* (although this term appears to be used more often when the original graph is connected). A *nonseparable* graph is a nontrivial connected graph with no articulation points. A *block* (or *bicomponent* or *biconnected component*) is a maximal nonseparable subgraph of a graph. If a block has order at least 3, it has vertex connectivity at least two. The blocks of a graph can intersect at articulation points. An articulation point always belongs to at least two blocks (possibly more), and any pair of intersecting blocks intersect at precisely one articulation point.

There are similarities and differences between components, strong components, and blocks. Thus, the components of a graph are the maximal connected parts of a graph; while the blocks of a graph are its maximal internally two vertex connected parts (except for those that happen to degenerate to a single edge). The components of a graph G partition both the vertices and edges of G into disjoint sets; while the blocks partition only the edges of G into disjoint sets, since the blocks themselves can intersect at articulation points. The strong components of a digraph, in contrast, partition the vertices, though not necessarily the edges of the digraph, since not every edge need belong to a strong component, unless it lies on a cycle.

We will present an algorithm for finding the blocks of a graph which is based on depth first search. The algorithm is similar to the strong components algorithm. The block algorithm is based on the following theorem. The parameter Low referred to in the theorem is defined in the same way as the parameter Low used in the orientation algorithm. We leave the proof of the theorem as an exercise.

THEOREM (BLOCK/ARTICULATION POINT STRUCTURE) Let G(V,E) be a graph. Let r be a vertex in G, and let T(r) be the depth first search tree rooted at r. Let a be an articulation point of G and let B be a block of G containing a. Then, all the vertices in B (except a) lie in a subtree of T rooted at a child w of a. The root r is an articulation point if and only if it has more than one child; while a <> r is an articulation point if and only if it has a child w such that Low(w) ò Dfsnum(a).

The block procedure follows. The data types used in the procedure are similar to those used for the strong component and orientation algorithms. Of course, the graph G is undirected. The search path is SP. The BCL stack stores the block vertices in a nested manner and is similar to the CL stack used for the strong components algorithm. The outputs of the procedure are the sets of vertices in each block. The sets are listed as the blocks are recognized.

```
Procedure Block(G)

(* Lists the blocks of G *)

var  G: Graph
     SP, BCL: Stack Entry pointer
     u, v, p: 1..|V|
     Nextcount, Predecessor_of_Top: Integer function
     Next, Empty: Boolean function
```

The proof of the claim is by contradiction. Suppose on the contrary that there exists a vertex other than v for which Low and Dfsnum are equal. Let u be the first such vertex popped from the depth first stack, and consider the depth first search subtree T(u) rooted at u. (Refer to Figure 5-17.) Since G is strongly connected, there exists a directed path Q(u,v) in G from u to v. Let w be the last vertex on Q(u,v) which is contained in T(u), and let z be the first vertex after w on Q(u,v).

Since z is not in T(u), the search process must have reached z before u. Therefore, Dfsnum(z) must be less than Dfsnum(u). Furthermore, since u is the first vertex for which the Low and Dfsnum values are equal, the only condition under which the component list stack is ever popped, z must still be on the component list stack when (w,z) is scanned. Therefore, the strong connectivity procedure will set Low(w) to at least as small as Dfsnum(z). Since w lies in the subtree T(u), the algorithm then eventually sets Low(u) to at least as small as Dfsnum(z), contradicting the assumption that Low(u) equals Dfsnum(u). The claim follows from this contradiction.

To complete the proof of the theorem in the special case, we observe that since every vertex in G is reached by the search tree before the search path retracts from v, every vertex in G must still be on the component list stack when the algorithm retracts from v and lists the members of the strong component recognized at that point. These will be precisely all the vertices in G. Thus, the algorithm correctly recognizes the sole strong component in G when G is strongly connected.

Figures 5-17 and 5-18 here

The argument in the general case of theorem proceeds as follows. The search starts at the vertex v in some strong component of the digraph and advances until it enters a strong component containing no outgoing edge to any other strong component. By the very same argument that we used in the special case, the algorithm will correctly recognize this strong component and remove its vertices from the component stack list. Once identified, that strong component will be subsequently ignored by the algorithm, and its vertices can never enter the component list stack again. Indeed, the search procedure acts from this point as if it were scanning a digraph lacking that strong component, and so the result follows by induction. This completes the proof of the theorem.

Refer to Figure 5-15 for an example. The search starts at component C2 and advances through strong component C3. After the algorithm has identified C3, the C3 vertices are subsequently ignored. The same procedure is repeated until every strong component reachable from the entry vertex v has been identified. Observe that the strong components are listed in the same order as a depth first traversal of the condensation D would pop the vertices of D from a depth first stack.

BLOCK ALGORITHM

The components of a graph are its most primitive global structural feature. The blocks of a graph represent a refinement of this feature. Recall that a cut-vertex of a graph G(V,E) is a vertex whose removal increases the number of

```
repeat

   (* ADVANCE *)

   while   Next(Top(SP),u)  do

       if    Dfsnum(u) =  0

       then  (* Advance *)
             Push(SP,u); Push(CL,u)
             Set Dfsnum(u) to Nextcount
             Set Low(u)    to Dfsnum (u)

       else  (* Compare *)
             if   u î CL  and*  Dfsnum(Top(SP)) ò Dfsnum(u)
             then Set Low(Top(SP)) to
                                min{Low(Top(SP)), Dfsnum(u)}

   (* BACK UP *)

   if     Dfsnum (Top(SP)) = Low (TOP(SP))

   then   (* Display strong component *)
          Pop and list CL up to and including Top(SP)

   else   (* Inherit *)
          Set p = Predecessor_of_Top(SP)
          Set Low(p)  to  min {Low(p), Low(Top(SP))}

until EMPTY (POP(SP))

End_Procedure_Strong
```

Before establishing the correctness of the algorithm, it will be convenient to introduce the *condensation digraph* D associated with a digraph G(V,E). Corresponding to each strong component s of G, we define a vertex s' in V(D). There is an edge from s' to u' in V(D) if and only if there is a (similarly directed) edge between the strong components corresponding to s' and u' in G. Figure 5-18 shows the condensation of the digraph of Figure 5-14.

THEOREM (CORRECTNESS OF STRONG COMPONENTS ALGORITHM) Let G(V,E) be a directed graph, and let v be any vertex in G. Then, the strong components algorithm finds all the strong components in G reachable from v.

The proof is as follows. We will first establish the correctness of the algorithm in the special case where the digraph G is strongly connected, that is, consists of a single strong component. We will then extend the proof to the case of arbitrary digraphs. Thus, let us first assume that G(V,E) is strongly connected, and let v be the root vertex used for the depth first search. We will show that in this case:

Claim: For every vertex u <> v, Low(u) < Dfsnum(u).

v_1,v_2,v_5,v_6,v_7	v_1,v_2,v_3,v_5,v_6,v_7	Scan (v_7,v_8) Push v_8	v_8:8	v_8:8
v_1,v_2,v_5,v_6,v_7,v_8	$v_1,v_2,v_3,v_5,v_6,v_7,v_8$	Scan (v_8,v_6)		v_8:6
v_1,v_2,v_5,v_6,v_7,v_8	$v_1,v_2,v_3,v_5,v_6,v_7,v_8$	Pop v_8		v_7:6
v_1,v_2,v_5,v_6,v_7	$v_1,v_2,v_3,v_5,v_6,v_7,v_8$	Pop v_7		
v_1,v_2,v_5,v_6	$v_1,v_2,v_3,v_5,v_6,v_7,v_8$	Pop v_6 from path. Display and Pop$\{v_6,v_7,v_8\}$ from list.		
v_1,v_2,v_5	v_1,v_2,v_3,v_5	Pop v_5		v_2:1
v_1,v_2		Pop v_2		
v_1	v_1,v_2,v_3,v_5	Pop v_1 from path. Display and Pop$\{v_1,v_2,v_3,v_5\}$ from list.		
Empty	Empty	Exit.		

Figure 5-16. Trace of Strong Components Algorithm from v_1.

```
Procedure Strong(G,v)

(* Finds and lists the strong components of G reachable from v *)

var  G: Graph
     SP, CL: Stack Entry pointer
     u, v, p: 1..|V|
     Nextcount, Top, Predecessor_of_Top: Integer function
     Next, Empty: Boolean function
     Pop: Stack Entry pointer function

Set Dfsnum(v) to Nextcount
Set Low(v)    to Dfsnum(v)
Create(SP); Push (SP,v)
Create(CL); Push (CL,v)
```

(4) Cross edges may lie either within a strong component (between vertices of the component) or between strong components. In the latter case, the algorithm ignores the edge in calculating Low (which is essentially used for intracomponent purposes) by recognizing that the terminal vertex v is not on the CL stack. On the other hand, when v is still on the CL stack, both u and v must lie in the same strong component; so the algorithm treats (u,v) like a back edge.

A formal statement of the procedure for finding strong components follows. The data types are basically the same as for the orientation algorithm. We can package an indicator array in CL to support efficient CL membership testing. A trace of the operation of the algorithm for the graph of Figure 5-14 is shown in Figure 5-16. The strong components are shown in Figure 5-15. Observe that only the strong components C1, C2, and C3 are reachable from v_1.

SEARCH PATH	COMPONENT LIST	ACTION	Dfsnum	Low
v_1	v_1	Initialization	v_1:1	v_1:1
v_1	v_1	Scan (v_1,v_2) Push v_2	v_2:2	v_2:2
v_1,v_2	v_1,v_2	Scan (v_2,v_3) Push v_3	v_3:3	v_3:3
v_1,v_2,v_3	v_1,v_2,v_3	Scan (v_3,v_2)		v_3:2
v_1,v_2,v_3	v_1,v_2,v_3	Pop v_3		
v_1,v_2	v_1,v_2,v_3	Scan (v_2,v_4) Push v_4	v_4:4	v_4:4
v_1,v_2,v_4	v_1,v_2,v_3,v_4	Display {v_4} and Pop v_4 from path. Pop list up to v_4.		
v_1,v_2	v_1,v_2,v_3	Scan (v_2,v_5) Push v_5	v_5:5	v_5:5
v_1,v_2,v_5	v_1,v_2,v_3,v_5	Scan (v_5,v_1) Scan (v_5,v_6) Push v_6	 v_6:6	v_5:1 v_6:6
v_1,v_2,v_5,v_6	v_1,v_2,v_3,v_5,v_6	Scan (v_6,v_7) Push v_7	v_7:7	v_7:7

adjacent strong component, the search continues into that adjacent component, and explores it fully, before returning to and completing the exploration of the earlier strong component. This behavior allows us to traverse the strong components in a nested fashion, while identifying them sequentially.

Like the orientation algorithm, the strong components algorithm uses a parameter *Low* which aids in the recognition of the strong components in a manner similar to the recognition of bridges in the orientation algorithm. That is, the critical points occur where the search retracts from vertices v where   Dfsnum(v) = Low(v).

The basic data structures are a *search stack* SP and a so-called *component list stack* CL. SP is handled in the usual depth first manner. The CL vertices correspond to vertices from partially traversed strong components. As the search progresses, it saves vertices on CL so that at any point CL contains every vertex reached since the last complete strong component was identified, as well as any previously reached vertices from strong components that have so far been only partially traversed. (Remember the depth first traversal makes the search enter the strong components in a nested manner, so that at any point CL contains fragments of many strong components.) Vertices are added to CL in tandem with SP as the search extends. Unlike SP, CL is popped only at those intermittent points at which a strong component is identified. This occurs at vertices v which satisfy the condition Dfsnum(v) = Low(v) which indicates the traversal of a strong component has been completed. The vertices on CL from Top(CL) up to and including v lie in the identified component and are popped.

The strong components algorithm does not explicitly use the digraph edge classification, but it is implicit. The different types of edges are treated as follows.

(1) We handle tree edges in the usual manner, using them to advance the search tree. A tree edge may or may not lead to a new strong component; however, the first search edge into a strong commponent is always a tree edge.

(2) We treat back edges as in the orientation algorithm. If (u,v) is a back edge, then u and v must be in the same strong component, since the tree path from v to u together with the back edge from u to v determine a cycle. Consequently, every vertex on this cycle is in the same strong component.

(3) Forward edges have no effect on our knowledge of the strong components. If (u,v) is a forward edge, and u and v happen to lie in the same component, the value of Low(u) will automatically incorporate the value of Low(v), because v lies in the search subtree under u, regardless of whether (u,v) is an edge or not. On the other hand, if u and v happen to lie in different components, the algorithm will have fully explored the strong component which v lies in before it completes exploring the strong component u lies in, by virtue of the depth-first character of the traversal. Accordingly, by the time (u,v) is scanned, the vertices of v will no longer be on the CL stack. The algorithm utilizes this phenomenon by ignoring any previously reached vertices no longer on CL. Thus, as in the other case, we effectively ignore the forward edge.

Nontrapping Condition: If Low(v) < Dfsnum(v) for every vertex v not equal to the root r, then G is orientable.

Figure 5-9 and 5-10 here

We can show conversely that if there is some subtree T(v) (other than T itself) for which v is trapped in T(v), then v is incident with a bridge, namely the entry edge to the subtree T(v). The corresponding trapping condition follows. Par(v) denotes the parent vertex of v in the depth first tree.

Trapping Condition: If Low(v) = Dfsnum(v) for some vertex v not equal to the root r, then (par(v),v) is a bridge of G.

To prove this condition, suppose Low(v) = Dfsnum(v) for some v not equal to r, and suppose that (par(v),v) is not a bridge. Refer to Figure 5-11 for an illustration. Then, there must be a back edge (u, w) from some vertex u in T(v) to an ancestor w of v. Consequently, Low(v) must be at least as small as Dfsnum(w), by the back edge and inheritance rules. Therefore, Low(v) is less than Dfsnum(v), contrary to assumption.

Figure 5-11 here

This completes the proof of the correctness of the conditions. We summarize the results in a theorem.

THEOREM (TRAPPING/NON-TRAPPING CONDITION FOR ORIENTABILITY) Let G(V,E) be a connected graph, and let r be the root of a depth first search tree in G. Then, if Low(v) < Dfsnum(v) for every vertex v not equal to r, then G is orientable. On the other hand, if Low(v) = Dfsnum(v) for some vertex v not equal to r, the edge (par(v),v) is a bridge of G.

It is now easy to design a depth first search procedure to compute Low and test the trapping condition. Refer to Figures 5-12 and 5-13 for an example.
Figures 5-12 and 5-13 here

5-4 STRONG COMPONENTS ALGORITHM

We define a *strong component* of a digraph G(V,E) as a connected induced subgraph of G of maximal order. The strong components of a digraph are analogous to the components of an undirected graph, corresponding to the maximal internally connected pieces of the digraph. See Figures 5-14 and 5-15 for an example. We present an algorithm for finding the strong components of a digraph based on depth first search. The algorithm uses an enhanced depth first search, similar to the orientation algorithm.

Figures 5-14 and 5-15 here

The depth first organization of the algorithm is critical to its correct operation. It guarantees that whenever the search enters a region of the graph corresponding to a strong component and which has an exit edge to an can show that if G is orientable, then G' is an orientation of G; while if G is not orientable, the individual components of the graph G'', obtained from G by removing its bridges, are oriented.

We can decide which case has occurred by considering the induced depth first directed tree T rooted at r in G'. G' is an orientation of G if and only if

(1) There is a directed path in T from r to every vertex in G', and

(2) There is a directed path from every vertex in G' to r.

The first condition is automatically satisfied when G is connected, since T is a directed tree with root r. Therefore, we need only consider when (2) holds. We consider this question using a "trapping condition" we will define.

Let v be a vertex in G', and let T(v) be the subtree of T rooted at v. We will say a vertex x in T(v) is "trapped in T(v)" if there is no directed path in G' from x to any vertex outside T(v). If not all vertices in T(v) are trapped in T(v), there must be a back edge from at least one vertex u in T(v) to some vertex w not in T(v). Since T is a depth first search tree, w must be an ancestor of v. Thus, if any vertex in T(v) is not trapped in T(v), there must be a directed path in G' from v to some vertex with a lower Dfsnum than v.

We will say a subtree T(v) satisfies a nontrapping condition if none of its vertices are trapped in T(v). If every subtree (except T itself, of course) satisfies a nontrapping condition, there is a directed walk from any vertex in G' to the root r of T. The walk is identified by merely repeatedly following the directed paths guaranteed to exist from each vertex to an ancestor vertex, until we eventually reach the root r, the vertex of least Dfsnum. Since, a directed walk between a pair of vertices necessarily contains a directed path between those vertices, then under these circumstances condition (2) holds. We refer to Figure 5-8 for an illustration.

Figure 5-8 here

The nontrapping condition motivates introducing, for every vertex v in G, a parameter Low(v) given by

min { Dfsnum(w) | u is in T(v) and (u,w) is a back edge }.

The interpretation of Low(v) is that it equals the Dfsnum of the earliest ancestor of v which is reachable from v by a directed path in T(v) followed by a single back edge. In other words, it directly implements the nontrapping condition. We can compute Low(v) (refer to Figures 5-9 and 5-10) by combining a back edge rule which sets Low(v) to min {Low(v), Dfsnum(u)}, where the minimum is taken over all back edges (v,u) at v, and an inheritance rule which sets Low(v) to min {Low(v), Low(u)}, where the minimum is taken over all children u of v. The back edge rule has the effect of identifying the earliest ancestor reachable from a vertex in G' using only a single back edge. The inheritance rule includes in Low(v) the least Dfsnum reachable by any vertex in the subtree T(v). Using Low, we can then formulate the nontrapping condition as:

5-3 ORIENTATION ALGORITHM

n undirected graph G(V,E) is *orientable* if we can impose directions on its edges E(G) (determining a new set of directed edges E') in such a way that the digraph G'(V,E') is strongly connected. The digraph G' is called an *orientation* of the undirected graph G. Refer to Figure 5-6 for examples. If we think of G as a roadmap whose edges correspond to two-way roads, G is orientable if and only if there is some way in which we can make every road one-way and yet still be able to travel between any pair of points in G.

Figure 5-6 here

We will describe two algorithms for testing whether a graph is orientable. One approach uses depth first search, and is more efficient. The other is a straightforward application of the following graph-theoretic characterization of orientability. Recall that a *bridge* is an edge whose removal increases the number of components in a graph.

THEOREM (ORIENTABILITY CHARACTERIZATION) A connected graph G(V,E) is orientable if and only if it contains no bridges.

This theorem implies that nonorientable connected graphs are precisely the graphs of edge connectivity one, and therefore an algorithm for testing orientability also tests whether EC(G) is at most one. The necessity of the condition in the theorem is obvious. If the graph contains a bridge, it cannot be orientable because whatever direction is assigned to the bridge will force access to be blocked in the opposite direction. On the other hand, it is not obvious that the absence of a bridge ensures orientability. This will follow from our proof of correctness of the depth first search based orientation algorithm.

The exhaustive search algorithm suggested by the theorem is outlined in Figure 5-7. Its performance is determined by the frequency of execution of its search loop, which depends both on 3E(G)3 and on whether the graph is orientable or not. If the graph is orientable, every edge must be examined before the algorithm terminates. If the graph is not orientable, the algorithm terminates as soon as a bridge is detected. Thus, in the worst case, the search loop is executed O(|E|) times. The connectivity test embedded in the loop takes O(|E|) time using depth first search, hence the overall complexity of the algorithm is O(|E|®^{2}).

Figure 5-7 here

DEPTH-FIRST ORIENTATION

We now describe an algorithm based on depth first search that determines orientability in O(|E|) steps. This algorithm also finds an orientation if one exists, in contrast to the exhaustive search algorithm which merely tests for orientability. The algorithm is based on the following idea. Any depth first search traversal of an undirected (connected) graph G induces a directed graph G'. The tree edges of the traversal are directed in G' from lower to higher Dfsnum, and the back edges are directed from higher to lower Dfsnum. We

```
                **else**  (* Cross edge *)
                **Set** Edge Type (u,v) to Cross

Pop(S)

**End_Procedure**_DFS_EC
```

ALTERNATIVE CLASSIFICATION ALGORITHMS

The previous depth first classification algorithm uses an explicit stack. There are alternatives. One is to use an array of switches, denoted Completed (|V|), which is initially set to false for every vertex and is set to true for a vertex v once all the edges at v are explored, or equivalently, when the search path retracts from v. The switches can be used to differentiate between back and cross edges. We have:

THEOREM (SWITCH CHARACTERIZATION OF CROSS AND BACK EDGES) Let G(V,E) be a directed graph. If an edge (x,y) of G is being explored during the execution of a depth first traversal of G, then (x,y) is a cross edge if and only if

    Dfsnum(y) < Dfsnum(x)   **and**   Completed(y) is true.

The test in this theorem must be applied during the execution of the depth first search, since once the search is done, Completed(y) will be true for every vertex y. The associated traversal algorithm is straightforward.

Another alternative is based on distinguishing between back and forward edges based on the order of completion of traversal of vertices. Recall that the variable Dfsnum ranks vertices according to the order in which they are added to the search path stack or first explored. We can define a complementary variable Completion which ranks vertices according to the order in which they are popped from the stack. Thus, while the first vertex added to the search path is assigned Dfsnum equal to 1, the first vertex the search path backs up from would be assigned Completion equal to 1. We assume that Completion(v) is initialized to some large integer M (which acts as plus infinity), for every vertex v.   This ensures the Completion number of incompletely traversed vertices is greater than the Completion number of completely traversed vertices. We have the following theorem.

THEOREM (ANCESTOR CHARACTERIZATION USING COMPLETION-NUM) Let G(V,E) be a directed graph. Let (u,v) be an edge in G. Then,

(1) (u,v) is a back edge with respect to a given depth first search if and only if Dfsnum(v) < Dfsnum(u) and Completion(u) ó Completion(v).

(2) (u,v) is a cross edge with respect to a given depth first search if and only if Dfsnum(v) < Dfsnum(u) and Completion(v) < Completion(u).

The depth-first search algorithm based on this theorem is straightforward.

THEOREM (CYCLE DETECTION BY DEPTH FIRST SEARCH) Let G(V,E) be a directed graph, and suppose the edges of G have been classified with respect to a depth first search traversal. Then, G is acyclic if and only if no edge of G has been classified as a back edge.

The proof is as follows. Clearly, if there is a back edge, G is not acyclic. Suppose, on the other hand, that G contains a cycle with successive vertices v_1 ,..., v_n. Let v_i be the first cycle vertex reached during the depth first search. Assume for simplicity that i > 1. Then, vi-1 must be a descendant of v_i in the depth first search tree. Therefore, the cycle edge (v_{i-1}, v_i) must be a back edge with respect to the depth first search, as was to be shown.

The algorithm that classifies the edges follows. We will assume the Stack S is represented as an array, augmented by an indicator array On. On(v) is set to 1 if v is on the stack, and to 0 otherwise. Pop is used as a procedure.

```
type  Stack = record
               SS(|V|): 1..|V|
               Top, On(|V|): 0..|V|
              end


Procedure DFS_EC(G,S,u)

(* DFS digraph edge classification with explicit stack *)

var  G: Graph
     u, v, w: 1..|V|
     S: Stack
     Nextcount: Integer function
     Next: Boolean function

Set Dfsnum(u) to Nextcount
Push(S,u)

while Next(u,v) do

   if    Dfsnum(v) = 0

   then  (* Tree edge *)
         Set Edge Type (u,v) to Tree
         DFS_EC(G,S,v)

   else  if Dfsnum(v) > Dfsnum(u)

         then  (* Forward edge *)
               Set Edge Type (u,v) to Forward

         else  if v is on search path stack S

               then  (* Back edge*)
               Set Edge Type (u,v) to Back
```

first traversal as

(1) A *tree edge* if y is unexplored when (x,y) is explored from x,

(2) A *back edge* if y is an ancestor of x with respect to the depth first tree,

(3) A *forward edge* if y is a descendant of x in the depth first tree, or

(4) A *cross edge* if neither x nor y is a descendant of one another.

Not every vertex may be reachable from a given root u. Therefore, several depth first searches, each starting from a different vertex, may be needed to explore the whole graph. Each search determines a corresponding (disjoint) depth first tree. Except for cross edges, the edges are always between vertices of a single depth first tree. A cross edge may connect either a pair of vertices within a single tree or a vertex in one search tree to a vertex in a previously traversed search tree. A procedure for classifying the edges is

(1) If y is unexplored, (x,y) is a tree edge, by definition;

(2) Otherwise, if Dfsnum(y) > Dfsnum(x), (x,y) must be a forward edge; since y cannot be an ancestor of x, (x,y) is not a back edge; and (x,y) cannot be a cross edge since Dfsnum(y) exceeds Dfsnum(x).

(3) Otherwise, (x,y) is a back or cross edge, depending on whether y is an ancestor of x or not; if y is still on the search path when explored from x, then y is an ancestor of x; so (x,y) is a back edge. Otherwise, (x,y) is a cross edge.

Figure 5-4 shows how to classify an edge (x,y) being explored from a vertex x, and Figure 5-5 gives an example. These results are summarized in the next theorem.

Figure 5-4 and 5-5 here

THEOREM (CLASSIFICATION OF EDGES FOR DIGRAPHS) Let G(V,E) be a directed graph and suppose the edges of G have been classified with respect to some depth first search traversal. Let (x,y) be an edge of G. Then,

(1) Dfsnum(x) < Dfsnum(y) if and only if (x,y) is a tree or forward edge, and

(2) Dfsnum(x) > Dfsnum(y) if and only if (x,y) is a back or cross edge.

The edge classification can be used to determine structural properties of a graph, as illustrated by the following.

```
         DFS_EC(G,S,v)

   else  if v = Predecessor_of_Top(S)
         then  (* Tree edge, 2®nd visit *)
               Set Edge Type (u,v) to Tree
         else  (* Back edge, either visit *)
               Set Edge Type (u,v) to Back
Pop(S)
End_Procedure_DFS_EC
```

The explicit stack is used only to identify the predecessor of Top(S). We can avoid the stack by retaining the top two vertices of the search path at each invocation. The implicit recursion stack will still maintain the complete search path, which is required for backtracking. The revised procedure DFS_EC(G,w,u) follows. The procedure is initially invoked as DFS_EC(G,nil,u).

```
Procedure DFS_EC(G,w,u)

(*  Recursive DFS from u with DFS search predecessor w *)

var  G: Graph
     u, v, w: 1..|V|
     Nextcount, Top: Integer function
     Next: Boolean function

Set Dfsnum(u) to Nextcount

while Next(u,v) do

   if    Dfsnum (v) = 0


   then  (* Tree edge, 1st visit *)
         Set Edge Type (u,v) to Tree
         DFS_EC(G,u,v)

   else  if v = w

         then  (* Tree edge, 2nd visit *)
               Set Edge Type (u,v) to Tree

         else  (* Back edge *)
               Set Edge Type (u,v) to Back

End_Procedure_DFS_EC
```

5-2-3 EDGE CLASSIFICATION: DIRECTED GRAPHS

The classification of edges for directed graphs is more diverse, but since the edges are represented nonredundantly, the classification is automatically consistent. We classify a directed edge (x,y) with respect to a given depth

```
Create(S)
Set Dfsnum(u) to Nextcount
Push (S,u)

repeat

   while Next(Top(S),v) do

      if    Dfsnum(v) = 0

      then  (* Tree edge, 1®st visit *)
             Set Edge Type (u,v) to Tree
             Set Dfsnum(v) to Nextcount
             Push  (S,v)


      else  if   v = Predecessor_of_Top(S)

            then  (* Tree edge, 2®nd visit *)
                  Set Edge Type (u,v) to Tree

            else  (* Back edge, either visit *)
                  Set Edge Type (u,v) to Back

until Empty(Pop(S))

End_Procedure_DFS_EC
```

We will now present a recursive version of the edge classification algorithm. The algorithm, DFS_EC, uses an explicit stack S to differentiate between a tree edge on its second visit and a back edge. This explicit stack was unnecessary in the pure vertex numbering algorithm because the distinction was not required then. We will assume S is initially empty.

```
Procedure DFS_EC(G,S,u)

(* Recursive DFS edge classification from u: undirected graph *)

var  G: Graph
     S: Stack Entry pointer
     u, v: 1..|V|
     Nextcount, Top, Predecessor_of_Top: Integer function
     Next: Boolean function
Set Dfsnum(u) to Nextcount
Push (S,u)

while Next(u,v) do

   if    Dfsnum(v) = 0

   then  (* Tree edge, 1®st visit *)
         Set Edge Type (u,v) to Tree
```

Figure 5-3 here

Back edges satisfy the following important property.

THEOREM (BACK EDGE PROPERTY FOR UNDIRECTED GRAPHS) Let G(V,E) be an undirected graph, and let T be an induced depth first search tree on G. Let (u,v) be a back edge with respect to T. Then:

(1) The vertices u and v stand in an ancestral/descendant relationship in T.

(2) If Dfsnum(u) < Dfsnum(v), u is an ancestor of v; otherwise v is an ancestor of u.

(3) The edge (u,v) is first explored from whichever of the vertices is the descendant vertex.

We can use the edge classification to induce a direction on the edges of G as follows:

(1) The tree edges are directed from tree parent to tree child;

(2) The back edges are directed from descendant to ancestor, or equivalently, from the vertex with the higher Dfsnum to the vertex with the lower Dfsnum.

This defines an *induced digraph* which we will consider further when we examine the problem of graph orientability.

The following algorithm classifies the edges of an undirected graph G. We modify our previous Edge data type to

```
type  Edge  = record
                Neighbor: 1..|V|
                Edge Type: (Tree, Back)
                Successor: Edge pointer
              end
```

A function Predecessor_of_Top returns the value of Top(Pop(S)), or zero if there are less than two elements on the stack.

```
Procedure DFS_EC(G,u)

(* Depth-first search edge classification: undirected graphs *)

var  G: Graph
     S: Stack Entry pointer
     u, v: 1..|V|
     Nextcount, Top, Predecessor_of_Top: Integer function
     Next, Empty: Boolean function
     Pop: Stack Entry pointer function
```

directed, and to an induced direction when G is undirected. We can easily augment any of the procedures for performing a depth first search so they also actually construct the depth first search tree. The critical structural feature of depth first search is given by the following theorem.

THEOREM (DEPTH FIRST SEARCH STRUCTURE THEOREM) Let G(V,E) be a graph (digraph) on which a depth first traversal is performed, creating a directed tree T(u) rooted at an initial vertex u. Let x and y be any pair of vertices reachable from u. Let T(x) and T(y) denote the subtrees of T(u) rooted at x and y respectively. If Dfsnum(x) < Dfsnum(y),

(1) Either y is in T(x), or

(2) T(x) and T(y) are disjoint, and there is no edge in G from T(x) to T(y).

It is this feature that makes depth first searching useful in identifying the structure of graphs, especially those properties related to connectivity. The breadth-first search satisfies no similar structural property, though we can still define a breadth-first search induced tree as analogous to the depth-first search induced tree.

5-2-2 EDGE CLASSIFICATION: UNDIRECTED GRAPHS

The applications of depth-first search depend on the classification of edges that it leads to. The classifications differ depending on whether the graph is directed or not. We will consider undirected graphs first.

We will classify the edges of undirected graphs into two types: tree edges and back edges. The *tree edges* are the edges in the induced depth-first search tree. Recall that we said an edge (x,y) was an edge of the induced tree if y was unexplored when we explored (x,y) from x. For consistency, (x,y) ought to be classified the same whether viewed from x or from y. By definition, a *back edge* is any edge which is not a tree edge. Once again, consistency demands that both representatives of a back edge be classified identically.

Since G is undirected, we must take care that redundantly represented undirected edges are classified uniquely. That is, since the distinct representatives of an edge refer to the same undirected edge, they ought to be classified identically in any consistent classification scheme.

The traversal examines each undirected edge twice, once at each representative of the edge. We can distinguish between tree and back edges by distinguishing between the first and second explorations of an undirected edge (x,y).

(1) Tree edge, first visit
(2) Tree edge, second visit
(3) Back edge, first visit
(4) Back edge, second visit

Figure 5-3 shows a scheme for classifying an edge (x,y).

(Q2) Enqueue(Q,v)
(Q3) Head(Q)
(Q4) Dequeue(Q)
(Q5) Empty(Q)

Their definitions are analogous to the corresponding stack operations. The algorithm is as follows.

```
Procedure BFS(G,u)

(* Breadth-first traversal of vertices in G reachable from u *)

var  G: Graph
     u, v: 1..|V|
     Nextcount: Integer function
     Next, Empty: Boolean function
     Q: Queue entry pointer
     Head, Dequeue: Queue entry pointer function

Create(Q)
Set Bfsnum(u) to Nextcount
Enqueue(Q,u)


repeat

  (* Number every unvisited vertex adjacent to Head(Q) *)

  while  Next(Head(Q),v) do

    if Bfsnum(v) = 0 then  Set Bfsnum(v) to Nextcount
                           Enqueue(Q,v)

until  Empty(Dequeue(Q))  (* Retreat or Exit *)

End_Procedure_BFS
```

A breadth-first search numbering of the vertices of the graph in Figure 5-2a is shown in Figure 5-2c. The depth-first labels are shown in Figure 5-2b. Refer to the exercises for a procedure that determines the breadth first search level of a vertex as well.

Figure 5-2 here

INDUCED DEPTH FIRST SEARCH TREE

The depth first traversal from a vertex u in a graph (or digraph) G implicitly determines a directed tree T rooted at u which we call the *depth first search tree*. V(T) consists of every vertex reachable from u. An edge (x,y) is in E(T) if y was as yet unexplored when (x,y) was explored from x. We consider (x,y) as directed from x to y, making T a directed tree rooted at u. This corresponds to the built-in direction of the edge when G is

```
until  Empty(Pop(S))  (* Retract Search Path or Exit *)

End_Procedure_DFS
```

RECURSIVE DEPTH FIRST SEARCH

We can also specify the depth-first search procedure recursively. This simplifies the procedure because the search path stack is maintained automatically and implicitly as part of the recursion. Each activation of the recursive procedure corresponds to an advance of the search path, while each return from a recursive activation corresponds to a retraction of the search path. This makes the control logic for the backtracking features of the algorithm transparent. The recursive DFS procedure is as follows.

```
Procedure DFS(G,u)

(* Performs recursive depth-first search from u *)

var  G: Graph
     u, v: 1..|V|
     Nextcount: Integer function
     Next: Boolean function

Set Dfsnum(u) to Nextcount

while  Next(u,v)  do   if  Dfsnum(v) = 0  then   DFS(G,v)

End_Procedure_DFS
```

The argument u may be either a value (that is, input only) parameter, or a reference (that is, input and output) parameter. The variable v in the procedure should be declared as a local variable. Then, each time DFS invokes itself, a new instance of v is allocated, with its own storage local to that activation. The search path stack is implicitly stored by the recursion mechanism as the sequence of such allocations and so is transparent. If we make u a value parameter, recursion also provides a stack in which the successive values of u are saved, corresponding to a second (redundant) copy of the search path stack.

BREADTH FIRST SEARCH

If we change the data structure used to store vertices from a stack to a queue, we obtain breadth-first search. Breadth-first search visits vertices in a level by level fashion, nearer vertices first. For example, if the graph is a tree, Breadth- first search visits the root (at level zero) first, then the vertices at level one (in some order), and so on. For an arbitrary graph, the vertices at level i of the search are precisely the vertices at distance i from the initial vertex.

The algorithm uses the standard queue operations.

(Q1) Create(Q)

```
                    end
```

The depth-first search procedure calls a variety of functions and other procedures. We use standard *stack operations*.

(S1) Create(S)
(S2) Push(S,v)
(S3) Top(S)
(S4) Pop(S)
(S5) Empty(S)

Create(S) sets S to a nil stack entry pointer. Push(S,v) creates a stack entry for v and pushes it on the stack S. Top(S) returns in Top the index of the next vertex on the stack. Pop(S) deletes the top of the stack S and also returns a pointer to the new top in Pop. Empty(S) succeeds when the stack is empty, and fails otherwise.

The function Next(x,y) returns in y the index of the next neighbor of vertex x and fails if there is none. Nextcount is an auto-increment integer function which is initially zero.

The control logic of the procedure mirrors the advance and retraction of the search path. The **while** loop advances the search path until it is blocked at a vertex all of whose neighbors have been explored by the time we reach the vertex. The outer loop retracts the search path by backing it up to the previous vertex on the search path stack. The search terminates when the search path is empty. Figure 5-1 gives a flowchart for the algorithm.

Figure 5-1 here

```
Procedure DFS(G, u)

(* Performs DFS traversal of vertices in G reachable from u,
   where G is either an undirected or directed graph *)

var  G: Graph
     S: Stack Entry pointer
     u, v: 1..|V|
     Nextcount, Top: Integer function
     Next, Empty: Boolean function
     Pop: Stack Entry pointer function

Create (S)
Set Dfsnum(u) to Nextcount
Push (S, u)

repeat

  while  Next (Top(S), v) do

    if Dfsnum(v) = 0  then  (* Advance Search Path *)
                            Set Dfsnum (v) to Nextcount
                            Push (S, v)
```

undirected graphs, the edge classification is simple. For directed graphs, the classification is more complex. In subsequent sections, we will show how the edge classifications can be used to analyze the structure of graphs.

5-2-1 VERTEX NUMBERING

We will first informally describe depth first search. The procedure is the same for both undirected and directed graphs.

(1) We start the search at a vertex v and initialize the search path to v.

(2) If there is an unexplored edge at the vertex at the head of the search path leading to an unexplored vertex w, we extend the search path to w, which becomes its new head, and repeat step (2).

(3) If there is an unexplored edge at the head of the search path leading to an explored vertex w, we mark that edge as explored and repeat (2) for the next edge incident at the head of the search path.

(4) If there are no further unexplored edges at the head of the search path, we retract the search path to the previous vertex by removing the current head of the path and repeat step (2).

We stop once the search path is empty.

The name depth-first search derives from the pattern of the search path which always advances from its last explored vertex, thus making the path drive deeper into the graph before it explores closer regions of the graph.

We will now describe a formal procedure for depth-first search. We represent the graph or digraph as a linear array H(|V|) with components of type Vertex.

```
type  Vertex = record
                 Dfsnum: 0..|V|
                 Positional Pointer: Edge pointer
                 Successor: Edge pointer
               end
      Edge   = record
                 Neighbor: 1..|V|
                 Successor: Edge pointer
               end
```

Dfsnum holds the depth-first search number of the vertex and is initially zero. As usual, the positional pointer for x points to the last explored edge at x. We call the graph (digraph) data type Graph. We use a stack S to represent the search path. Its entries have the following form.

```
type  Stack Entry = record
                      Index: 1..|V|
                      Successor: Stack Entry pointer
```

® DEPTH FIRST SEARCH

5-1 INTRODUCTION

We traverse a data structure by visiting all its nodes and pointers. For example, there are several standard techniques for traversing a binary tree. If the tree is a binary search tree, we use inorder traversal to list its search keys in order. If the tree represents an expression in infix form, we use preorder traversal to convert the expression to prefix form.

Graph traversal techniques vastly generalize these ideas. We consider arbitrary graphs instead of ordered binary trees, and we typically assume no special ordering for the edges incident at a vertex of the graph. Graph traversal procedures show how to systematically explore a graph, both to access information stored at its vertices and edges, and to identify structural properties such as whether or not the graph is connected.

Depending on how a graph is represented, it may or may not be easy to even identify its vertices and edges. For example, if a graph is represented as a linear array, we can directly access its vertices and sequentially access its edge lists, although even so simple a property as connectedness will not be obvious from the representation. On the other hand, if a graph is represented by a pure linked representation, where we must follow pointers to access the graph, it requires care even to find all the vertices and edges in the graph.

The two most common techniques for traversing graphs are depth- first search and breadth-first search. In *depth-first search*, we repeatedly extend a search path as far as possible into a graph, retract it, and then reextend it in another direction, until we have traversed the whole graph. In breadth-first search, on the other hand, we explore the graph in a level by level fashion, starting the search at a given vertex, and progressively extending the search to vertices at greater distances.

Depth-first search is useful in identifying structural properties of graphs. On the other hand, for problems such as maximum flow (which we will consider in Chapter 6), where the structural characteristics of depth-first search are not needed, breadth- first search may lead to better performance.

Despite their different graphical interpretations, the implementations of the algorithms for depth- and breadth-first searches are quite similar, differing only in the data structure used to control the search. Depth-first search uses a stack for this purpose, while breadth-first search uses a queue. As a consequence, depth-first search can be implemented naturally using recursion. Our discussion will concentrate on depth-first search and some of its variations.

5-2   DEPTH FIRST SEARCH ALGORITHMS

We will initially describe depth-first traversal in its simplest form, where we number vertices according to the order in which they are visited without classifying edges. We will then augment the algorithm to classify edges. For