CHAPTER 3

Depth-First Search and Applications*

Krishnaiyan “KT” Thulasiraman

CONTENTS

3.1 Introduction ......................................................... 59
3.2 DFS of an Undirected Graph ...................................... 59
3.3 DFS of a Directed Graph ........................................... 63
3.4 Biconnectivity and Strong Connectivity Algorithms ............ 66
   3.4.1 Biconnectivity Algorithm .................................... 66
   3.4.2 Strong Connectivity Algorithm ................................ 68
3.5 \( s-t \)-Numbering of a Graph ..................................... 71

3.1 INTRODUCTION

In this chapter we describe a systematic method for exploring a graph. This method known as depth-first search, in short, DFS, has proved very useful in the design of several efficient algorithms [2,3]. We describe three of these application, namely, biconnectivity, strong connectivity and \( s-t \) numbering algorithms.

3.2 DFS OF AN UNDIRECTED GRAPH

We first describe DFS of an undirected graph. To start with, we assume that the graph under consideration is connected. If the graph is not connected, then DFS would be performed separately on each component of the graph. We also assume that there are no self-loops in the graph. DFS of an undirected graph \( G \) proceeds as follows:

We choose any vertex, say \( v \), in \( G \) and begin the search from \( v \). The start vertex \( v \), called the root of the DFS, is now said to be visited.

We then select an edge \( (v, w) \) incident on \( v \) and traverse this edge to visit \( w \). We also orient this edge from \( v \) to \( w \). The edge \( (v, w) \) is now said to be examined and is called a tree edge. The vertex \( v \) is called the father of \( w \), denoted as FATHER(\( w \)).

In general, while we are at some vertex \( x \), two possibilities arise:

1. If all the edges incident on \( x \) have already been examined, then we return to the father of \( x \) and continue the search from FATHER(\( x \)). The vertex \( x \) is now said to be completely scanned.

2. If there exists some unexamined edges incident on \( x \), then we select one such edge \( (x, y) \) and orient it from \( x \) to \( y \). The edge \( (x, y) \) is now said to be examined. Two cases need to be considered now:

*This chapter is an edited version of Sections 11.7, 11.8, and 11.10 in Thulasiraman and Swamy [1].
Case 1 If \( y \) has not been previously visited, then we traverse the edge \((x, y)\), visit \( y \), and continue the search from \( y \). In this case \((x, y)\) is a root edge and \( x = \text{FATHER}(y) \).

Case 2 If \( y \) has been previously visited, then we proceed to select another unexamined edge incident on \( x \). In this case the edge \((x, y)\) is called a back edge.

During the DFS, whenever a vertex \( x \) is visited for the first time, it is assigned a distinct integer \( \text{DFN}(x) \) such that \( \text{DFN}(x) = i \), if \( x \) is the \( i \)th vertex to be visited during the search. \( \text{DFN}(x) \) is called the depth-first number (DFN) of \( x \). Clearly, DFNs indicate the order in which the vertices are visited during DFS.

DFS terminates when the search returns to the root and all the vertices have been visited.

We now present a formal description of the DFS algorithm. In this description the graph under consideration is assumed to be connected. The array \( \text{MARK} \) used in the algorithm has one entry for each vertex. To begin with we set \( \text{MARK}(v) = 0 \) for every vertex \( v \) in the graph, thereby indicating that no vertex has yet been visited. Whenever a vertex is visited for the first time, we set the corresponding entry in the \( \text{MARK} \) array equal to 1. We use an array \( \text{SCAN} \) that has one entry for each vertex in the graph. To begin with we set \( \text{SCAN}(v) = 0 \) for every vertex \( v \), thereby indicating that none of the vertices is completely scanned. Whenever a vertex is completely scanned, the corresponding entry in the \( \text{SCAN} \) array is set to 1. The arrays \( \text{DFN} \) and \( \text{FATHER} \) are as defined before. \( \text{TREE} \) and \( \text{BACK} \) are two sets storing, respectively, the tree edges and the back edges as they are generated.

Algorithm 3.1 DFS of an undirected graph

**Input:** \( G = (V, E) \) is a connected undirected graph. Vertex \( s \) is the start vertex of the depth-first search.

**Output:** Depth-first numbering of the vertices of \( G \) and the depth-first search tree with vertex \( s \) as the root vertex.

**begin**

\[ \text{TREE} \leftarrow \emptyset; \]
\[ \text{BACK} \leftarrow \emptyset; \]

\[ \text{for every edge } e \text{ in } G, \text{EXAMINED}(e) \leftarrow 0; \]

\[ \text{for vertex } v \text{ in } G \]
\[ \quad \text{do} \]
\[ \quad \text{FATHER}(v) \leftarrow v; \]
\[ \quad \text{MARK}(v) \leftarrow 0; \]
\[ \quad \text{SCAN}(v) \leftarrow 0; \]

\[ \text{od} \]
\[ \text{MARK}(s) \leftarrow 1; \]
\[ \text{DFN}(s) \leftarrow 1; \]
\[ i \leftarrow 1; \]
\[ v \leftarrow s; \]

**repeat**

\[ \text{while there exists an edge } e = (v,w) \text{ with EXAMINED}(e) = 0 \]
\[ \quad \text{do} \]
\[ \quad \text{Orient the edge } (v,w) \text{ from } v \text{ to } w; \]
\[ \quad \text{EXAMINED}(e) \leftarrow 1; \]
\[ \quad \text{if } \text{MARK}(w) = 0 \text{ then} \]
\[ \quad \quad \text{begin} \]
\[ \quad \quad i \leftarrow i + 1; \]

\[ \text{end} \]

\[ \text{end} \]

"
Depth-First Search and Applications  ■  61

Depth-First Search and Applications

As we can see from the preceding description, DFS partitions the edges of \( G \) into tree edges and back edges. It is easy to show that the tree edges form a spanning tree of \( G \). DFS also imposes directions on the edges of \( G \). The resulting directed graph will be denoted by \( \hat{G} \). The tree edges with their directions imposed by the DFS will form a directed spanning tree of \( \hat{G} \). This directed spanning tree will be called the DFS tree.

Note that DFS of a graph is not unique since the edges incident on a vertex may be chosen for examination in any arbitrary order.

As an example, we have shown in Figure 3.1 DFS of an undirected graph. In this figure tree edges are shown as continuous lines, and back edges are shown as dashed lines. Next to each vertex we have shown its DFN. We have also shown in the figure the list of edges incident on each vertex \( v \). This list for a vertex \( v \) is called the adjacency list of \( v \), and it gives the order in which the edges incident on \( v \) are chosen for examination.

Let \( T \) be a DFS tree of a connected undirected graph. As we mentioned before, \( T \) is a directed spanning tree of \( G \). For further discussions, we need to introduce some terminology.

If there is a directed path in \( T \) from a vertex \( v \) to a vertex \( w \), then \( v \) is called an ancestor of \( w \), and \( w \) is called a descendant of \( v \). Furthermore, if \( v \neq w \), \( v \) is called a proper ancestor of \( w \), and \( w \) is called a proper descendant of \( v \). If \( (v, w) \) is a directed edge in \( T \), then \( v \) is called the father of \( w \), and \( w \) is called a son of \( v \). Note that a vertex may have more than one son. A vertex \( v \) and all its descendants form a subtree of \( T \) with vertex \( v \) as the root of this subtree.

Two vertices \( v \) and \( w \) are related if one of them is a descendant of the other. Otherwise \( v \) and \( w \) are unrelated. If \( v \) and \( w \) are unrelated and \( \text{DFN}(v) < \text{DFN}(w) \), then \( v \) is said to be to the left of \( w \); otherwise, \( v \) is to the right of \( w \). Edges of \( G \) connecting unrelated vertices are called cross edges. We now show that there are no cross edges in \( G \).

Let \( v_1 \) and \( v_2 \) be any two unrelated vertices in \( T \). Clearly then there are two distinct vertices \( s_1 \) and \( s_2 \) such that (1) \( \text{FATHER}(s_1) = \text{FATHER}(s_2) \) and (2) \( v_1 \) and \( v_2 \) are descendants of \( s_1 \) and \( s_2 \), respectively (see Figure 3.2).

Let \( T_1 \) and \( T_2 \) denote the subtrees of \( T \) rooted at \( s_1 \) and \( s_2 \), respectively. Assume without loss of generality that \( \text{DFN}(s_1) < \text{DFN}(s_2) \). It is then clear from the DFS algorithm that vertices in \( T_2 \) are visited only after the vertex \( s_1 \) is completely scanned.

Further, scanning of \( s_1 \) is completed only after all the vertices in \( T_1 \) are scanned completely. So there cannot exist an edge connecting \( v_1 \) and \( v_2 \). For if such an edge existed, it would have been visited before the scanning of \( s_1 \) is completed.

### Algorithm for DFS

```plaintext
DFN(w) ← i;
TREE ← TREE ∪ \{(v, w)\};
MARK(w) ← 1;
FATHER(w) ← v;
v ← w;
end
else BACK = BACK ∪ \{(v, w)\};
end while
SCAN(v) ← 1;
v ← FATHER(v);
until v = s and SCAN(s) = 1;
end
```
Handbook of Graph Theory, Combinatorial Optimization, and Algorithms

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Adjacency List</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1, 2), (1, 3), (1, 4)</td>
</tr>
<tr>
<td>2</td>
<td>(2, 1), (2, 3), (2, 8), (2, 9), (2, 10), (2, 11)</td>
</tr>
<tr>
<td>3</td>
<td>(3, 1), (3, 2), (3, 4), (3, 5), (3, 6), (3, 7)</td>
</tr>
<tr>
<td>4</td>
<td>(4, 1), (4, 3), (4, 5), (4, 6)</td>
</tr>
<tr>
<td>5</td>
<td>(5, 3), (5, 4)</td>
</tr>
<tr>
<td>6</td>
<td>(6, 3), (6, 4)</td>
</tr>
<tr>
<td>7</td>
<td>(7, 3), (7, 8)</td>
</tr>
<tr>
<td>8</td>
<td>(8, 2), (8, 7)</td>
</tr>
<tr>
<td>9</td>
<td>(9, 2), (9, 10), (9, 11)</td>
</tr>
<tr>
<td>10</td>
<td>(10, 2), (10, 9)</td>
</tr>
<tr>
<td>11</td>
<td>(11, 2), (11, 9)</td>
</tr>
</tbody>
</table>

Figure 3.1 DFS of an undirected graph.

Figure 3.2 Illustration of the proof of Theorem 3.1.
Thus we have the following.

**Theorem 3.1** If \((v, w)\) is an edge in a connected undirected graph \(G\), then in any DFS tree of \(G\) either \(v\) is a descendant of \(w\) or vice versa. In other words, there are no cross edges.

The absence of cross edges in an undirected graph is an important property that forms the basis of an algorithm to be discussed in Section 3.4.1 for determining the biconnected components of a graph.

### 3.3 DFS OF A DIRECTED GRAPH

DFS of a directed graph is essentially similar to that of an undirected graph. The main difference is that in the case of a directed graph an edge is traversed only along its orientation. As a result of this constraint, edges in a directed graph \(G\) are partitioned into four categories (and not two as in the undirected case) by a DFS of \(G\). An unexamined edge \((v, w)\) encountered while at the vertex \(v\) would be classified as follows.

**Case 1** \(w\) has not yet been visited.

In this case \((v, w)\) is a tree edge.

**Case 2** \(w\) has already been visited.

a. If \(w\) is a descendant of \(v\) in the DFS forest (i.e., the subgraph of tree edges), then \((v, w)\) is called a **forward edge**.

b. If \(w\) is an ancestor of \(v\) in the DFS forest, then \((v, w)\) is called a **back edge**.

c. If \(v\) and \(w\) are not related in the DFS forest and \(DFN(w) < DFN(v)\), then \((v, w)\) is a **cross edge**. Note that there are no cross edges of the type \((v, w)\) with \(DFN(w) > DFN(v)\). The proof for this is along the same lines as that for Theorem 3.1.

A few useful observations are now in order:

1. An edge \((v, w)\), with \(DFN(w) > DFN(v)\), is either a tree edge or a forward edge. During the DFS it is easy to distinguish between a tree edge and a forward edge because a tree edge always leads to a new vertex.

2. An edge \((v, w)\) with \(DFN(w) < DFN(v)\) is either a back edge or a cross edge. Such an edge \((v, w)\) is a back edge if and only if \(w\) is not completely scanned when the edge is encountered while examining the edges incident out of \(v\).

3. DFS forest, the subgraph of tree edges, may not be connected even if the directed graph under consideration is connected. The first vertex to be visited in each component of the DFS forest will be called the root of the corresponding component.

A description of the DFS algorithm for a directed graph is presented next. As we pointed out earlier, when we encounter an edge \((v, w)\) with \(DFN(w) < DFN(v)\), we shall classify it as a back edge if \(SCAN(w) = 0\); otherwise \((v, w)\) is a cross edge. We also use two arrays, FORWARD and CROSS, that store respectively, forward and cross edges.
Algorithm 3.2 DFS of a directed graph

*Input:* $G = (V, E)$ is a connected directed graph.  
*Output:* Depth first numbering of the vertices of $G$.

```
begin 
    TREE ← ϕ; BACK ← ϕ; FORWARD ← ϕ; CROSS ← ϕ;
    for every edge $e$ in $G$ EXAMINED $(e) ← 0$;
    for every vertex $v$ in $G$
        do
            FATHER $(v) ← v$; MARK $(v) ← 0$; SCAN $(v) ← 0$;
            ROOT $(v) ← 0$;
        od
    i ← 0;
    repeat
        while there exists a vertex $v$ with MARK $(v) = 0$;
            MARK $(v) ← 1$; $i ← i + 1$; DFN $(v) ← i$; ROOT $(v) ← 1$;
        repeat
            while there exists an edge $e ← (v, w)$ with EXAMINED $(e) = 0$
                do
                    Orient the edge $(v, w)$ from $v$ to $w$; EXAMINED $(e) ← 1$;
                    if MARK $(w) ← 0$ then
                        begin
                            $i ← i + 1$;
                            DFN $(w) ← i$;
                            TREE ← TREE ∪ {(v, w)};
                            MARK $(w) ← 1$;
                            FATHER $(w) ← v$;
                            $v ← w$;
                        end
                    else if DFN $(w) >$ DFN $(v)$ then
                        FORWARD ← FORWARD ∪ {(v, w)};
                    else if SCAN $(w) = 0$ then
                        BACK ← BACK ∪ {(v, w)};
                    else CROSS ← CROSS ∪ {(v, w)};
                od
            end while
            SCAN $(v) ← 1$;
            $v ←$ FATHER $(v)$;
        until ROOT $(v) = 1$ and SCAN $(v) = 1$
    end while
    until $i = n$;
end
```

As an example, DFS of a directed graph is shown in Figure 3.3a. Next to each vertex we have shown its DFN. The tree edges are shown as continuous lines, and the other edges are shown as dashed lines. The DFS forest is shown separately in Figure 3.3b.

We pointed out earlier that the DFS forest of a directed graph may not be connected, even if the graph is connected. This can also be seen from Figure 3.3b. This leads us to the problem of discovering sufficient conditions for a DFS forest to be connected. In the following we prove that the DFS forest of a strongly connected graph is connected. In fact, we shall be
establishing a more general result. Let $T$ denote a DFS forest of a directed graph $G = (V, E)$. Let $G_i = (V_i, E_i)$, with $|V_i| \geq 2$, be a strongly connected component of $G$. Consider any two vertices $v$ and $w$ in $G_i$. Assume without loss of generality that $DFN(v) < DFN(w)$. Since $G_i$ is strongly connected, there exists a directed path $P$ in $G_i$ from $v$ to $w$. Let $x$ be the vertex on $P$ with the lowest DFN and let $T_x$ be the subtree of $T$ rooted at $x$. Note that cross edges and back edges are the only edges that lead out of the subtree $T_x$. Since these edges lead to vertices having lower DFNs than $DFN(x)$, it follows that once path $P$ reaches a vertex in $T_x$, then all the subsequent vertices on $P$ will also be in $T_x$. In particular, $w$ also lies in $T_x$. So it is a descendant of $x$. Since $DFN(x) \leq DFN(v) < DFN(w)$, it follows from the DFS algorithm that $v$ is also in $T_x$. Thus, any two vertices $v$ and $w$ in $G_i$ have a common ancestor that is also in $G_i$.

We may conclude from this that all the vertices of $G_i$ have a common ancestor $r_i$ that is also in $G_i$. It may now be seen that among all the common ancestors in $T$ of vertices in $G_i$ vertex $r_i$ has the highest DFN. Further, it is easy to show that if $v$ is a vertex in $G_i$, then any vertex on the tree path from $r_i$ to $v$ will also be in $G_i$. So the subgraph of $T$ induced by $V_i$ is connected. Thus we have the following.

**Theorem 3.2** Let $G_i = (V_i, E_i)$ be a strongly connected component of a directed graph $G = (V, E)$. If $T$ is a DFS forest of $G$, then the subgraph of $T$ induced by $V_i$ is connected.  ■
Following is an immediate corollary of Theorem 3.2.

**Corollary 3.1** *The DFS forest of a strongly connected graph is connected.*

It is easy to show that the DFS algorithms are both of complexity $O(n + m)$, where $n$ is the number of vertices and $m$ is the number of edges in a graph.

### 3.4 BICONNECTIVITY AND STRONG CONNECTIVITY ALGORITHMS

In this section we discuss algorithms due to Hopcroft and Tarjan [2] and Tarjan [3] for determining the biconnected components and the strongly connected components of a graph. These algorithms are based on DFS. We begin our discussion with the biconnectivity algorithm.

#### 3.4.1 Biconnectivity Algorithm

First we recall that a biconnected graph is a connected graph with no cut-vertices. A maximal biconnected subgraph of a graph is called a biconnected component of the graph.*

A crucial step in the development of the biconnectivity algorithm is the determination of a simple criterion that can be used to identify cut-vertices as we perform a DFS. Such a criterion is given in the following two lemmas.

Let $G = (V, E)$ be a connected undirected graph. Let $T$ be a DFS tree of $G$ with vertex $r$ as the root. Then we have the following.

**Lemma 3.1** *Vertex $v \neq r$ is a cut-vertex of $G$ if and only if for some son $s$ of $v$ there is no back edge between any descendant in $T$ of $s$ (including itself) and a proper ancestor of $v$.*

**Proof.** Let $G'$ be the graph that results after removing vertex $v$ from $G$. By definition, $v$ is a cut-vertex of $G$ if and only if $G'$ is not connected.

Let $s_1, s_2, \ldots, s_k$ be the sons of $v$ in $T$. For each $i$, $1 \leq i \leq k$, let $V_i$ denote the set of descendants of $s_i$ (including itself), and let $G_i$ be the subgraph of $G'$ induced on $V_i$. Further let $V'' = V' - \bigcup_{i=1}^{k} V_i$, where $V' = V\setminus \{v\}$, and let $G''$ be the subgraph induced on $V''$. Note that all the proper ancestors of $v$ are in $V''$.

Clearly, $G_1, G_2, \ldots, G_k$ and $G''$ are all subgraphs of $G$, which together contain all the vertices of $G'$. We can easily show that all these subgraphs are connected. Further, by Theorem 3.1 there are no edges connecting vertices belonging to different $G_i$’s. So it follows that $G'$ will be connected if and only if for every $i$, $1 \leq i \leq k$, there exists an edge $(a, b)$ between a vertex $a \in V_i$ and a vertex $b \in V''$. Such an edge $(a, b)$ will necessarily be a back edge, and $b$ will be a proper ancestor of $v$. We may therefore conclude that $G''$ will be connected if and only if for every son $s_i$ of $v$ there exists a back edge between some descendant of $s_i$ (including itself) and a proper ancestor of $v$. The proof of the lemma is now immediate. ■

**Lemma 3.2** *The root vertex $r$ is a cut-vertex of $G$ if and only if it has more than one son.*

**Proof.** Proof in this case follows along the same line as that for Lemma 3.1. ■

In the following we refer to the vertices of $G$ by their DFNs. To embed into the DFS procedure the criterion given in Lemmas 3.1 and 3.2, we now define, for each vertex $v$ of $G$,

$$\text{LOW}(v) = \min(\{v\} \cup \{w|\text{there exists a backedge } (x, w) \text{ such that } x \text{ is a descendant of } v, \text{ and } w \text{ is a proper ancestor of } v \text{ in } T\}) \quad (3.1)$$

*Note that a biconnected component is the same as a block defined in Chapter 1.*
Using the LOW values, we can restate the criterion given in Lemma 3.1 as in the following theorem.

**Theorem 3.3**  
Vertex \( v \neq r \) is a cut-vertex of \( G \) if and only if \( v \) has a son \( s \) such that \( \text{LOW}(s) \geq v \).

Noting that \( \text{LOW}(v) \) is equal to the lowest numbered vertex that can be reached from \( v \) by a directed path containing at most one back edge, we can rewrite (3.1) as

\[
\text{LOW}(v) = \min(\{v\} \cup \{\text{LOW}(s)| s \text{ is a son of } v\} \cup \{w|(v,w) \text{ is a backedge}\})
\]

This equivalent definition of \( \text{LOW}(v) \) suggests the following steps for computing \( \text{LOW}(v) \):

1. When \( v \) is visited for the first time during DFS, set \( \text{LOW}(v) \) equal to the DFN of \( v \).
2. When a back edge \((v, w)\) incident on \( v \) is examined, set \( \text{LOW}(v) \) to the minimum of its current value and the DFN of \( w \).
3. When the DFS returns to \( v \) after completely scanning a son \( s \) of \( v \), set \( \text{LOW}(v) \) equal to the minimum of its current value and \( \text{LOW}(s) \).

Note that for any vertex \( v \), computation of \( \text{LOW}(v) \) ends when the scanning of \( v \) is completed.

We next consider the question of identifying the edges belonging to a biconnected component. For this purpose we use an array \( \text{STACK} \). To begin with \( \text{STACK} \) is empty. As edges are examined, they are added to the top of \( \text{STACK} \).

Suppose DFS returns to a vertex \( v \) after completely scanning a son \( s \) of \( v \). At this point computation of \( \text{LOW}(s) \) will have been completed. Suppose it is now found that \( \text{LOW}(s) \geq v \). Then, by Theorem 3.3, \( v \) is a cut-vertex. Further, if \( s \) is the first vertex with this property, then we can easily see that the edge \((v, s)\) along with the edges incident on \( s \), and its descendants will form a biconnected component. These edges are exactly those that lie on top of \( \text{STACK} \) up to and including \((v,s)\). They are now removed from \( \text{STACK} \). From this point on the algorithm behaves in exactly the same way as it would on the graph \( G' \), which is obtained by removing from \( G \) the edges of the biconnected component that has just been identified.

For example, a DFS tree of a connected graph may be as in Figure 3.4, where \( G_1, G_2, \ldots, G_5 \) are the biconnected components in the order in which they are identified.

A description of the biconnectivity algorithm now follows. This algorithm is essentially the same as Algorithm 3.1, with the inclusion of appropriate steps for computing \( \text{LOW}(v) \) and identifying the cut-vertices and the edges belonging to the different biconnected components. Note that in this algorithm the root vertex \( r \) is treated as a cut-vertex, even if it is not one, for the purpose of identifying the biconnected component containing \( r \).

![Figure 3.4](image-url)  
*Figure 3.4  \( G_1, G_2, G_3, G_4, G_5 - \)biconnected components of a graph.*
Algorithm 3.3 Biconnectivity

**Input:** $G = (V, E)$ is a connected undirected graph.  
**Output:** Biconnected components of $G$.

```
begin  
STACK ← ϕ;  
for every edge $e$ in $G$ EXAMINED $(e) ← 0$;  
for vertex $v$ in $G$ do  
   FATHER $(v) ← v$;  
   MARK $(v) ← 0$;  
   SCAN $(v) ← 0$;  
od  
Pick any vertex $s$ with MARK $(s) ← 0$;  
MARK $(s) ← 1$;  
DFN $(s) ← 1$;  
LOW $(s) ← 1$;  
i ← 1;  
v ← s;  
repeat  
   while there exists an edge $e = (v, w)$ with EXAMINED $(e) = 0$ do  
      EXAMINED $(e) ← 1$;  
      STACK ← STACK ∪ $\{(v, w)\}$;  
      if MARK $(w) ← 0$ then  
         begin  
            MARK $(w) ← 1$;  
            i ← i + 1;  
            DFN $(w) ← i$;  
            FATHER $(w) ← v$;  
            LOW $(w) ← i$;  
            v ← w;  
         end  
      else LOW $(v) ← \min \{LOW (v), DFN (w)\}$;  
   od  
while  
SCAN $(v) ← 1$;  
if LOW $(v) ≥ DFN (FATHER (v))$ then  
   remove all the edges from the top of the STACK up to and including the edge (FATHER $(v)$, $v$);  
   LOW (FATHER $(v)$) ← \min \{LOW $(v)$, LOW (FATHER $(v)$)\};  
   v ← FATHER $(v)$;  
until v = s and SCAN $(s) = 1$;  
end
```

### 3.4.2 Strong Connectivity Algorithm

Recall from Chapter 1 that a graph is strongly connected if for every pair of vertices $v$ and $w$ there exists in $G$ a directed path from $v$ to $w$ and a directed path from $w$ to $v$; further a maximal strongly connected subgraph of a graph $G$ is called a strongly connected component of the graph.
Consider a directed graph $G = (V, E)$. Let $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \ldots, G_k = (V_k, E_k)$ be the strongly connected components of $G$. Let $T$ be a DFS forest of $G$ and $T_1, T_2, \ldots, T_k$ be the induced subgraphs of $T$ on the vertex sets $V_1, V_2, \ldots, V_k$, respectively. We know from Theorem 3.2 that $T_1, T_2, \ldots, T_k$ are connected.

Let $r_i, 1 \leq i \leq k$, be the root of $T_i$. If $i < j$, then DFS terminates at vertex $r_i$ earlier than at $r_j$. Then we can see that for each $i < j$, either $r_i$ is to the left of $r_j$ or $r_i$ is a descendant of $r_j$ in $T$. Further $G_i, 1 \leq i \leq k$, would consist of those vertices that are descendants of $r_i$, but are in none of $G_1, G_2, \ldots, G_{i-1}$.

The first step in the development of the strong connectivity algorithm is the determination of a simple criterion that can be used to identify the roots of strongly connected components as we perform a DFS. The following observations will be useful in deriving such a criterion. These observations are all direct consequences of the fact that there exist no directed circuits in the graph obtained by contracting all the edges in each one of the sets $E_1, E_2, \ldots, E_k$.

1. There is no back edge of the type $(v, w)$ with $v \in V_i$ and $w \in V_j, i \neq j$. In other words all the back edges that leave vertices in $V_i$ also end on vertices in $V_i$.
2. There is no cross edge of the type $(v, w)$ with $v \in V_i$, and $w \in V_j, i \neq j$ and $r_j$ is an ancestor of $r_i$. Thus for each cross edge $(v, w)$ one of the following two is true:
   a. $v \in V_i$ and $w \in V_j$ for some $i$ and $j$ with $i \neq j$ and $r_j$ to the left of $r_i$.
   b. For some $i, v \in V_i$ and $w \in V_r$.

Assuming that the vertices of $G$ are named by their DFS numbers, we define for each $v$ in $G$, $LOWLINK(v) = \min\{\{v\} \cup \{w\} \mid$ there is a cross edge or a back edge from a descendant of $v$ to $w,$ and $w$ is in the same strongly connected component as $v\}$.

Suppose $v \in V_i$. Then it follows from the above definition that $LOWLINK(v)$ is the lowest numbered vertex in $V_i$ that can be reached from $v$ by a directed path that contains at most one back edge or one cross edge. From the observations that we have just made it follows that all the edges of such a directed path will necessarily be in $G_i$. As an immediate consequence we get

$$LOWLINK(r_i) = r_i \text{ for all } 1 \leq i \leq k. \quad (3.2)$$

Suppose $v \in V_i$ and $v \neq r_i$. Then there exists a directed path $P$ in $G_i$ from $v$ to $r_i$. Such a directed path $P$ should necessarily contain a back edge or a cross edge because $r_i < v$, and only cross edges and back edges lead to lower numbered vertices. In other words $P$ contains a vertex $w < v$. So for $v \neq r_i$, we get

$$LOWLINK(v) < v. \quad (3.3)$$

Combining (3.2) and (3.3) we get the following theorem, which characterizes the roots of the strongly connected components of a directed graph.

**Theorem 3.4** A vertex $v$ is the root of a strongly connected component of a directed graph $G$ if and only if $LOWLINK(v) = v.$

The following steps can be used to compute $LOWLINK(v)$ as we perform a DFS.

1. On visiting $v$ for the first time, set $LOWLINK(v)$ equal to the DFS number of $v$.
2. If a back edge $(v, w)$ is examined, then set $LOWLINK(v)$ equal to the minimum of its current value and the DFS number of $w$. 

© 2016 by Taylor & Francis Group, LLC
3. If a cross edge \((v, w)\) with \(w\) in the same strongly connected component as \(v\) is explored, set \(\text{LOWLINK}(v)\) equal to the minimum of its current value and the DFS number of \(w\).

4. When the search returns to \(v\) after completely scanning a son \(s\) of \(v\), set \(\text{LOWLINK}(v)\) to the minimum of its current value and \(\text{LOWLINK}(s)\).

To implement step 3 we need a test to check whether \(w\) is in the same strongly connected component as \(v\). For this purpose we use an array \(\text{STACK1}\) to which vertices of \(G\) are added in the order in which they are visited during the DFS. \(\text{STACK1}\) is also used to determine the vertices belonging to a strongly connected component.

Let \(v\) be the first vertex during DFS for which it is found that \(\text{LOWLINK}(v) = v\). Then by Theorem 3.4, \(v\) is a root and in fact it is \(r_1\). At this point the vertices on top of \(\text{STACK1}\) up to and including \(v\) are precisely those that belong to \(G_1\). Thus, \(G_1\) can easily be identified. These vertices are now removed from \(\text{STACK1}\). From this point on the algorithm behaves in exactly the same way as it would on the graph \(G'\), which is obtained by removing from \(G\) the vertices of \(G_1\).

As regards the implementation of step 3 in \(\text{LOWLINK}\) computation, let \(v \in V_i\) and let \((v, w)\) be a cross edge encountered while examining the edges incident on \(v\). Suppose \(w\) is not in the same strongly connected component as \(v\). Then it would belong to a strongly connected component \(G_j\) whose root \(r_j\) is to the left of \(r_i\). The vertices of such a component would already have been identified, and so they would no longer be on \(\text{STACK1}\). Thus, \(w\) will be in the same strongly connected component as \(v\) if and only if \(w\) is on \(\text{STACK1}\).

A description of the strong connectivity algorithm now follows. This is the same as Algorithm 3.2 with the inclusion of appropriate steps for computing \(\text{LOWLINK}\) values and for identifying the vertices of the different strongly connected components. We use in this algorithm an array \(\text{POINT}\). To begin with \(\text{POINT}(v) = 0\) for every vertex \(v\). This indicates that no vertex is on the array \(\text{STACK1}\). \(\text{POINT}(v)\) is set to 1 when \(v\) is added to \(\text{STACK1}\), and it is set to zero when \(v\) is removed from \(\text{STACK1}\). We also use an array \(\text{ROOT}\). \(\text{ROOT}(v) = 1\) if it is the root vertex of a tree in the DFS forest. For example, in Figure 3.3b, 1, 8, and 13 are root vertices.

---

**Algorithm 3.4 Strong connectivity**

**Input:** \(G = (V, E)\) is a connected directed graph.

**Output:** Strongly connected components of \(G\).

**begin**

\(\text{STACK1} \leftarrow \phi\).

\(\text{for every edge } e \text{ in } G, \text{EXAMINED}(e) \leftarrow 0;\)

\(\text{for every vertex } v \text{ in } G\)

\(\text{begin}\)

\(\text{FATHER}(v) \leftarrow v;\)

\(\text{MARK}(v) \leftarrow 0;\)

\(\text{SCAN}(v) \leftarrow 0;\)

\(\text{ROOT}(v) \leftarrow 0;\)

\(\text{POINT}(v) \leftarrow 0;\)

\(\text{end}\)

\(i \leftarrow 0;\)

\(\text{repeat}\)

\(\text{while there exists a vertex } v \text{ with } \text{MARK}(v) = 0;\)

\(\text{MARK}(v) \leftarrow 1; i \leftarrow i + 1; \text{DFN}(v) \leftarrow i; \text{ROOT}(v) \leftarrow 1;\)

---
LOWLINK(v) ← i; STACK1 ← STACK1 ∪ {v}; POINT(v) ← 1;
repeat
  while there exists an edge e ← (v, w) with EXAMINED(e) = 0 do
    EXAMINED(v) ← 1;
    if MARK(w) = 0 then
      begin
        i ← i + 1;
        DFN(w) ← i;
        MARK(w) ← 1;
        LOWLINK(v) ← i;
        FATHER(w) ← v;
        v ← w;
        STACK1 ← STACK1 ∪ {w};
        POINT(w) ← 1;
      end
    else
      if DFN(w) < DFN(v) and POINT(w) = 1 then
        LOWLINK(v) ← min{LOWLINK(v), DFN(w)}.
    od
  end while
SCAN(v) ← 1;
if LOWLINK(v) = DFN(v), then
  begin
    remove all the vertices from the top of STACK1 up to
    and including v;
    POINT(x) ← 0 for all such x removed from STACK1;
  end
  if ROOT(FATHER(v)) = 0 then
    LOWLINK(FATHER(v)) ← min{LOWLINK(FATHER(v)),
    LOWLINK(v)};
    v ← FATHER(v);
  until ROOT(v) = 1 and SCAN(v) = 1;
end while
until i = n;
end

See Figure 3.5 for an illustration of this algorithm. In this figure, LOWLINK values are
shown in parentheses. Strongly connected components are {3, 4, 5}, {6, 7, 8, 9, 10}, {2}, and
{1, 11, 12, 13}.

3.5 st-NUMBERING OF A GRAPH

In this section we present yet another application of DFS-computing an st-numbering of a
graph. For an application of s−t numbering, see Reference [4].

Given an n-vertex biconnected graph G = (V, E) and an edge (s, t) of G, a numbering
of the vertices of G is called an st-numbering of G if the following conditions are satisfied,
where g(v) denotes the corresponding st-number of vertex v:

1. For all v ∈ V, 1 ≤ g(v) ≤ n, and for u ≠ v, g(u) ≠ g(v).
2. g(s) = 1.
A graph $G$ and an $st$-numbering of $G$ are shown in Figure 3.6. Lempel et al. [4] have shown that for every biconnected graph and every edge $(s, t)$ there exists an $st$-numbering. The $st$-numbering algorithm to be discussed next is due to Even and Tarjan [5].

Given a biconnected graph $G$ and an edge $(s, t)$, the $st$-numbering algorithm of Even and Tarjan first performs a DFS of $G$ with $t$ as the start (root) vertex and $(t, s)$ as the first edge. In other words $DFN(t) = 1$, and $DFN(s) = 2$. Recall that $DFN(v)$ denotes the DFS
number of vertex $v$. During the DFS, the algorithm also computes, for each vertex $v$, its DFN, FATHER($v$) in the DFS tree, the low point LOW($v$), and identifies the tree edges and back edges.

Next the vertices $s$ and $t$ and the edge $(s,t)$ are marked old. All the other vertices and edges are marked new. An algorithm, called the path finding algorithm, is then invoked repeatedly (in an order to be described later) until all the vertices and edges are marked old.

The path finding algorithm when applied from an old vertex $v$ finds a directed path into $v$ or from $v$ and proceeds as follows.

### Algorithm 3.5 Path finding algorithm (applied from vertex $v$)

**S1.** Pick a new edge incident on $v$.

   i. If $(v, w)$ is a back edge (DFN($w$) < DFN($v$)), then mark $e$ old and HALT. 
   *Note:* The path consists of the single edge $e$.

   ii. If $(v, w)$ is a tree edge (DFN($w$) > DFN($v$)), then do the following:
       Starting from $v$ traverse the directed path that defined LOW($w$) and mark all edges and vertices on this path old. HALT.
   *Note:* The path here starts with the tree edge $(v, t)$ and ends in the vertex $u$ such that DFN($u$) = LOW($w$). This path has exactly one back edge.

   iii. If $(w, v)$ is a back edge (DFN($w$) > DFN($v$)) do the following:
       Starting from $v$ traverse the edge $(w, v)$ backward and continue backward along tree edges until an old vertex is encountered. Mark all the edges and vertices on this path old and HALT.
   *Note:* The path in this case is directed into $v$.

**S2.** If all the edges incident on $v$ are old HALT.

*Note:* The path produced is empty.

The following facts hold true after each application of the path finding algorithm. Note that the algorithm is always applied from an old vertex.

1. All ancestors of an old vertex are old too. This is true before the first application of the algorithm since $t$ is the only ancestor of $s$ and it is old. This property remains true after any one of the applicable steps of the algorithm.

2. When the algorithm is applied from an old vertex $v$, it either produces an empty path or it produces a path that starts at $v$, passes through new vertices and edges and ends at another old vertex. This is obvious when $(v, w)$ or $(w, v)$ is a back edge (cases [i] and [iii] of S1). This is also true when $(v, w)$ is a tree edge because in the biconnected graph $G$ the vertex $u$ defining LOW($w$) is an ancestor of $v$ and therefore $u$ is old.

Even and Tarjan’s $st$-numbering algorithm presented next uses a stack STACK that initially contains only $t$ and $s$ with $s$ on top of $t$. In this description we do not explicitly include the details of DFS. We also assume that to start with $t$, $s$ and the edge $(t, s)$ are old.

### Algorithm 3.6 $st$-Numbering (Even and Tarjan)

**S1.** $i \leftarrow 1$.

**S2.** Let $v$ be the top vertex on STACK. Remove $v$ from STACK. If $v = t$, then $g(v) \leftarrow i$ and HALT.
Theorem 3.5 Algorithm 3.5 computes an \textit{st}-numbering for every biconnected graph \( G = (V, E) \).

\textbf{Proof.} The following facts about the algorithm are easy to verify:

1. No vertex appears in more than one place on STACK at any time.

2. Once a vertex \( v \) is placed on STACK, no vertex under \( v \) receives a number until \( v \) does.

3. A vertex is permanently removed from STACK only after all edges incident on \( v \) become \textit{old}.

We now show that each vertex \( v \) is placed on STACK before \( t \) is removed. Clearly this is true for \( v = s \) because initially \( t \) and \( s \) are placed on STACK with \( s \) on top of \( t \).

Consider any vertex \( v \neq s, t \). Since \( G \) is biconnected, there exists a directed path \( P \) of tree edges from \( s \) to \( v \). Let \( P: s, u_1, u_2, \ldots, u_k = v \). Let \( m \) be the first index such that \( u_m \) is not placed on STACK. Since \( u_{m-1} \) is placed on STACK, \( t \) can be removed only after \( u_{m-1} \) is removed (fact 2), and \( u_{m-1} \) is removed only after all edges incident on \( u_{m-1} \), are \textit{old} (fact 3). So \( u_m \) must be placed on STACK before \( t \) is removed.

We need to show that the numbers assigned to the vertices are indeed \textit{st}-numbers. Since each vertex is placed on STACK and eventually removed, every vertex \( v \) gets a number \( g(v) \).

Clearly all numbers assigned are distinct. Also \( g(s) = 1 \) and \( g(t) = n \) because \( s \) is the first vertex and \( t \) is the last vertex to be removed. Every time a vertex \( v \neq s, t \) is placed on STACK, there is an adjacent vertex placed above \( v \) and an adjacent vertex placed below \( v \). By fact 2 the one above gets a lower number and the one below gets a higher number. 


Further Reading

A number of algorithms that use DFS as a building block have been reported in the literature. For example, see References [8–12]. See Chapter 27 for algorithms on program graphs that use DFS.

References


