

## 1 Depth-first Search

Let  $G = (V, E)$  be a directed or undirected graph. Given a vertex  $a \in V$ , depth-first search identifies all vertices  $v$  such that either  $v = a$  or there is a path from  $a$  to  $v$  in  $G$ . It does so by setting  $mark(v) = T$  for all such vertices, where initially,  $mark(v) = F$  for all vertices.

(Note that  $mark = F$  and  $mark = T$  correspond to colors white and gray, respectively, in the pseudocode in the textbook. The color black is not really needed in the code, it is used only to facilitate understanding the algorithm.)

```
DFS( $v$ )
1.   $mark(v) := T$ 
2.  for each vertex  $w \in Adj[v]$  do
3.      if  $mark(w) = F$  then
4.           $\pi(w) := v$ 
5.          DFS( $w$ )
      fi
    rof
    ( $color(v) := black$ )
end {procedure DFS}
```

```
Main Program
  for each  $u \in V$  do
     $mark(u) := F$ 
     $\pi(u) := NIL$ 
  rof

  DFS( $a$ )
end {Main Program}
```

### Bound on Running Time

Let  $|V| = n$  and  $|E| = m$ . For an analysis of the running time of this algorithm, we note the following:

0. The initialization in the Main Program takes time  $O(n)$ .

1. For each vertex  $v \in V$ ,  $dfs(v)$  is called at most once. The reason for this is that
  - (i) a call to  $dfs(v)$  is made only if  $mark(v) = F$ ,
  - (ii)  $mark(v)$  is set to  $T$  upon execution of  $dfs(v)$ , and
  - (iii)  $mark(v)$  is never re-set to  $F$ .
2. The time taken by the call  $dfs(v)$ , *outside of the time taken by recursive calls within  $dfs(v)$*  is  $1 + O(deg^+(v))$  (the out-degree of  $v$ ), since the only computation outside of the recursive calls is to set  $mark(v)$  to  $T$  and to check if  $mark(w) = F$  for each edge  $(v, w)$  that is outgoing from  $v$ .

By 0, 1, and 2, the running time of this algorithm is

$$O(n) + \sum_{v \in V} (1 + O(deg^+(v))) = O(n) + n + O(m) = O(m + n)$$

In deriving the above expression, we used the fact that the sum of the out-degrees of the vertices in  $G$  is  $m$ .

The above analysis also shows that the algorithm terminates.

### Correctness.

In the following, we will say that a vertex  $v$  is *unmarked* if  $mark(v) = F$ , and it is *marked* if  $mark(v) = T$ . Note that unmarked vertices are ‘white’ vertices and marked vertices are vertices that are either ‘gray’ or ‘black’ according to the pseudocode in the textbook.

**Theorem 1.1** *Consider a call to  $DFS(u)$  invoked during the execution of the main program. We claim that this call to  $DFS(u)$  makes a recursive call to  $DFS(v)$  iff there is a path of unmarked vertices from  $u$  to  $v$  at the time when  $DFS(u)$  is called.*

### Proof:

**Only if.** Let  $DFS(v)$  be called during the call to  $DFS(u)$ . Then, we show by induction on the number of recursive calls made, that there is a path of unmarked vertices from  $u$  to  $v$  at the time when  $DFS(u)$  is called.

The base case is when no recursive call is made, i.e.,  $v = u$ . The base case holds since there is a path of length 0 from  $u$  to  $u$ .

Assume inductively that the result holds for up to  $k - 1$  recursive calls, and let the call to  $DFS(v)$  be the  $k$ th recursive call within  $DFS(u)$ . Let this recursive call be made when examining the adjacency list of vertex  $w$ . Then, the call  $DFS(w)$  was made as the  $j$ th recursive call within  $DFS(u)$ , for some  $j < k$ . Hence by the inductive assumption there was a path  $p$  of white vertices from  $u$  to  $w$  at the time when  $DFS(u)$  was called. Then, the path

$p$ , followed by the edge  $(w, v)$  was a path of white vertices from  $u$  to  $v$  at the time when  $\text{DFS}(u)$  was called.

**If.** Let  $S$  be the set of unmarked vertices that are reachable from  $u$  when  $\text{DFS}(u)$  is called, and let  $|S| = k$ . Since vertex  $u$  is unmarked when  $\text{DFS}(u)$  is called,  $|S| \geq 1$ .

The proof is by induction on  $k$  with the following inductive assertion:

1. If  $|S| = k$  then there are exactly  $k$  calls to  $\text{DFS}$  made during the call to  $\text{DFS}(u)$  (including the call to  $\text{DFS}(u)$ ), one to each vertex in  $S$ .

**Base.**  $k = 1$ . Then, the only unmarked vertex reachable from  $u$  at the time  $\text{DFS}(u)$  is called is  $u$  itself. Hence every vertex in  $\text{Adj}[u]$  is marked, so when the **for** loop in step 2 is executed during the call to  $\text{DFS}(u)$ , no recursive call is made, and the only call to  $\text{DFS}$  is to  $\text{DFS}(u)$ . Hence the inductive assertion holds for  $k = 1$ .

**Induction Step.** Let the inductive statement be true when up to  $k - 1$  unmarked vertices are reachable from any given vertex  $v$  when  $\text{DFS}(v)$  is called, and let  $|S| = k$  when  $\text{DFS}(u)$  is called.

Let  $S' = S - \{u\}$ . Let us partition the vertices in  $S'$  into disjoint groups according to the earliest vertex in  $\text{Adj}[u]$  that lies on a path from  $u$  to that vertex. Let us call these disjoint sets as  $S_i$ ,  $1 \leq i \leq r$ , and the corresponding earliest vertices on  $\text{Adj}[u]$  as  $v_i$ ,  $1 \leq i \leq r$ . Note that  $v_1$  is the first unmarked vertex in  $\text{Adj}[u]$ , and  $\{v_1, \dots, v_r\}$  is a subset of the unmarked vertices in  $\text{Adj}[u]$  at the time when  $\text{DFS}(u)$  is called.

Since  $v_1$  is the first unmarked vertex on  $\text{Adj}[u]$ , the first recursive call within  $\text{DFS}(u)$  is  $\text{DFS}(v_1)$ . By construction, the set of vertices reachable from  $v_1$  at the time of this call using only unmarked vertices is  $S_1$ . Since  $u$  is not in  $S_1$ , we have  $|S_1| \leq k - 1$ , and by inductive assertion 1, there is one recursive call to  $\text{DFS}(w)$  for each  $w \in S_1$  during the call to  $\text{DFS}(v_1)$ .

Assume inductively that for all  $j \leq i$  for some  $i < r$ ,  $\text{DFS}(v_j)$  recursively calls  $\text{DFS}(w)$  for each  $w \in S_j$ . Now consider  $v_i$ . By construction,

- $v_i$  is a vertex in  $\text{Adj}[u]$  that was unmarked when  $\text{DFS}(u)$  was called;
- there is no path from  $u$  to  $v_i$  that passes through a  $v_j$ ,  $j < i$ , since, otherwise  $v_j$  would have been added to  $S_j$ ;
- $S_i$  contains those unmarked vertices that were reachable from  $u$  along a path of unmarked vertices when  $\text{DFS}(u)$  was called, and  $v_i$  was the earliest vertex in  $\text{Adj}[u]$  that lies on any such path from  $u$  to that vertex. *Hence,  $S_i$  is the set of unmarked vertices that are reachable from  $v_i$  along a path of unmarked vertices when  $\text{DFS}(v_i)$  is called.*

Since  $|S_i| < k$ , by the inductive assertion 1, a call to  $\text{DFS}(v_i)$  recursively calls  $\text{DFS}(w)$  on all vertices  $w \in S_i$  and thus makes  $|S_i|$  calls (including the original call to  $\text{DFS}(v_i)$ ). Thus, inductively, this property holds for every  $i$ ,  $1 \leq i \leq r$ .

Thus, the total number of calls to `DFS` made when `DFS(u)` is called is

$$1 + \sum_{i=1}^r |S_i| = 1 + |S'| = k$$

and `DFS(u)` makes a recursive call on exactly those vertices  $v$  for which there is a path of unmarked vertices from  $u$  to  $v$  at the time when `DFS(u)` is called.

□

As in `BFS`, we observe that the  $\pi$  pointers define a tree rooted at  $a$ . We also observe that since  $\pi(v) := u$  is set when `DFS(v)` is called, by Theorem 1.1

- $v$  is a descendant of  $u$  in the tree if and only if there is a path of unmarked vertices from  $u$  to  $v$  at the time when `DFS(u)` is invoked.

This is called the ‘White-path theorem’ in your textbook.

## 1.1 Depth-first search on the entire graph

Depth-first search is usually used in a main program that ensures that it is called on every vertex in the graph. For this we modify the main program slightly as follows:

```
DFS-ALL( $G = (V, E)$ )
  for each  $u \in V$  do
     $mark(u) := F$ 
     $\pi(u) := NIL$ 
  rof

  for each  $a \in V$  do
    if  $mark(a) := F$  then DFS(a)
  rof
end {DFS-ALL( $G = (V, E)$ )}
```

`DFS-ALL( $G = (V, E)$ )` constructs a tree for each connected component if  $G$  is undirected (and if  $G$  is directed, it constructs a collection of trees that span all vertices in  $V$ ).

Using Theorem 1.1 and the White-path theorem, it is not difficult to derive the following theorem given in the textbook, which makes use of the  $d$  and  $f$  values that are assigned in the `DFS-VISIT` pseudocode given in your textbook.

**Theorem 1.2** (*Parenthesis theorem, reworded*) When  $\text{DFS-ALL}(G = (V, E))$  is executed using  $\text{DFS-VISIT}$  in the textbook in place of  $\text{DFS}$ , for any two vertices  $u, v \in V$  with  $d[u] < d[v]$ , exactly one of the following two statements hold.

1. The intervals  $[d[u], f[u]]$  and  $[d[v], f[v]]$  are disjoint (i.e.,  $d[u] < f[u] < d[v] < f[v]$ ), and  $u$  nor  $v$  is a descendant of the other in the depth-first forest;
2.  $d[u] < d[v] < f[v] < f[u]$ , and  $v$  is a descendant of  $u$  in the depth-first search forest.

**Classification of edges.** The edges in  $G$  that are not identified as tree edges during a depth-first search are the *nontree edges* in  $G$  with respect to the depth-first search. These nontree edges have some important structure to them, which is summarized as follows:

- **Undirected graph  $G$ .** Every nontree edge  $(u, v)$  is a *back edge* where  $u$  and  $v$  satisfy an ancestor-descendant relationship in the dfs tree.
- **Directed graph  $G$ .** A nontree edge  $(u, v)$  can be of 3 types:
  - *back edge* if  $v$  is an ancestor of  $u$
  - *forward edge* if  $v$  is a descendant of  $u$
  - *cross edge* if  $u$  and  $v$  do not have an ancestor-descendant relationship; in this case the  $d$  and  $f$  values will correspond to part 1 of Theorem 1.2, with  $u$  and  $v$  interchanged, i.e.,  $d[v] < f[v] < d[u] < f[u]$ .

The above characterization leads to a simple algorithm for *topological sort* using depth-first search.

## 2 Topological Sort on a DAG

A *directed acyclic graph* (or *DAG*) is a directed graph that contains no directed cycle.

**Lemma 2.1** *If a directed graph  $G$  is a DAG then there is no back edge in  $G$  with respect to a depth-first search.*

(The above result is in fact, an if and only if statement, and the topological sort algorithm that we provide below will establish the ‘only if’ direction.)

A *topological sort* of a DAG  $G = (V, E)$  is a labeling  $t(v)$  of its vertices with distinct labels from 1 to  $n$  (where  $n = |V|$ ) such that,

$$\text{for every edge } (u, v) \in E, \quad t(u) < t(v)$$

Using depth-first search we can design a simple linear-time algorithm to perform a topological sort on an undirected graph  $G = (V, E)$

TOPOLOGICAL-SORT( $G = (V, E)$ )

1. DFS-ALL( $G = (V, E)$ )
2. **for** each  $v \in V$  **do**  $t(v) := n + 1 - f(v)$

**Lemma 2.2** *The  $t$  values returned by algorithm TOPOLOGICAL-SORT represent a topological sort of  $G$  if  $G$  is acyclic.*

**Proof:** To prove correctness it suffices to show that if  $G$  is acyclic, then for each edge  $(u, v)$  in  $G$ ,  $f(u) > f(v)$ , since in that case  $t(u) < t(v)$ .

We establish this by considering in turn the case when  $(u, v)$  is a tree edge, a forward edge, and a cross edge.

□

Another interesting property of a directed graph is strong connectivity and strongly connected components. A directed graph is *strongly connected* if for every pair of vertices,  $u, v$  there is a path from  $u$  to  $v$  and a path from  $v$  to  $u$ .

If  $G$  is not strongly connected, we can decompose  $G$  into its *strongly connected components* (*scc's*), where each strongly connected component is a maximal subgraph of  $G$  that is strongly connected. As in the case of connected components in an undirected graph, the vertex sets of strongly connected components of directed graph  $G$  partition the vertex set of  $G$ . However, there may be edges in  $G$  that do not lie in any strongly connected component. In particular, if  $G$  is a DAG, then each scc is a single vertex, and all edges lie outside the scc's of  $G$ .

It is not difficult to design a linear-time algorithm to determine if a graph  $G$  is strongly connected. Finding strongly connected components is more tricky, however one can design a linear-time algorithm for this problem using depth-first search. This is given in your textbook, but we will not go into it here.