Analysis of Algorithms I: Depth-First Search and Topological Sort

Xi Chen

Columbia University

We discuss the second strategy commonly used in the "generic" algorithm for reachability described in the last class:

1. set $R = \{s\}$
2. while there is an edge from R to $V - R$ do
3. let $(u, v) \in E$ be such an edge with $u \in R$, $v \in V - R$
4. set $R = R \cup \{v\}$ and $v.\pi = u$

The strategy is called Depth-first search (DFS): For each round, choose an edge (u, v) from R to $V - R$, where u is the newest vertex added to R. Similarly we say u discovers v in this round, and set the pointer $v.\pi$ to be u. See an example of DFS on Page 605.

In DFS, each vertex v has one of the following three colors:

1. White: not discovered yet
2. Gray: discovered but not finished yet
3. Black: finished

with the two words "discover" and "finish" to be defined more formally later. At the beginning, all vertices are white. Each vertex $v \in V$ has an attribute $v.\pi$ (set to be nil at the beginning), in which we will store the vertex $u \in V$ that discovers v.

DFS-Visit(G, u), where $u \in V$ satisfies u.color $=$ white:

1. change u.color from white to gray (just discovered)
2. for each $v \in \mathrm{adj}(u)$ do
3. if v.color $=$ white (not discovered yet)
4. set $v.\pi = u$ (u discovers v)
5. DFS-Visit(G, v) (call DFS-Visit to explore v)
6. change u.color from gray to black (finished)

It is clear that we change the color of u from white to gray at the beginning of DFS-Visit(G, u) (we say u is just discovered), and change it again to black at the end (we say u is finished). But why do we name this procedure "DFS-Visit" instead of "DFS"? In most of the applications of DFS, we need to keep calling DFS-Visit until we have discovered all vertices of G. And we reserve "DFS" for the latter procedure that makes calls to DFS-Visit. (Comparison to BFS: One application of BFS is to compute the shortest-path distances from a given source vertex $s \in V$. To this end, it suffices to make one call BFS(G, s). But to use BFS to compute the connected components of an undirected graph, then one also needs to keep calling BFS until all vertices are discovered.)

In DFS-Visit(G, u), we enumerate vertices $v \in \mathrm{adj}(u)$ (clearly it is better to use the list representation here, just like in BFS) not discovered yet, and make a recursive call DFS-Visit(G, v) to explore v. Upon the termination of DFS-Visit(G, u), we use E_π to denote the following set of edges:

$$(v.\pi, v), \quad \text{for all } v \in V \text{ such that } v.\pi \neq \text{nil}$$

Using a similar argument from the last class, it is easy to show that $E_\pi \subseteq E$ (why?) has no cycle (why?) and thus, is a tree rooted at u. We call it the Depth-first tree formed by DFS-Visit(G, u).

Check Figure 22.4 on Page 605 in the textbook. DFS-Visit(G, u) discovers u first, followed by v, y and x. When x is discovered, it has no white neighbor in $\mathrm{adj}(x)$ so we are finished with x; change it from gray to black; and backtracks to y, the vertex that discovered x. Similarly, none of y, v and u has any white neighbor in their adjacency lists and we are done. For this example,

$$E_\pi = \Big\{(u, v), (v, y), (y, x)\Big\}$$

clearly forms a tree rooted at u.

In most of the applications, we start with a graph $G = (V, E)$, directed or undirected; set v.color = white and $v.\pi$ = nil for all $v \in V$; and keep calling DFS until every vertex are discovered: DFS (G), where $G = (V, E)$ is either undirected or directed:

1. for each vertex $v \in V$ do
2.   set v.color = white and $v.\pi$ = nil
3. set time = 0
4. for each vertex $u \in V$ do
5.   if u.color = white then
6.   DFS (G, u)

Upon termination of DFS(G), it is clear that we have discovered every vertex $v \in V$ and v.color $=$ white (because DFS-Visit(G, u) changes u to black by the end, and we never touch a vertex again once it is black). It is also easy to check that the edges

$$E_\pi = \Big\{(v.\pi, v) : v \in V \text{ with } v.\pi \neq \text{nil}\Big\}$$

form a forest of several depth-first trees: Every $v \in V$ belongs to exactly one of the trees. We call it the depth-first forest formed by DFS(G). Note that in DFS(G), we maintain a global counter *time* to record the time we discover each vertex $v \in V$ (changed from white to gray) as well as the time we are finished with v (changed from gray to black). We update DFS-Visit(G, u) as follows:

DFS-Visit(G, u):

1. set time $=$ time $+ 1$ and $u.d =$ time
2. change u.color from white to gray (just discovered)
3. for each $v \in \mathrm{adj}(u)$ do
4.   if v.color $=$ white (not discovered yet)
5.   set $v.\pi = u$ (u discovers v)
6.   DFS-Visit(G, v) (call DFS-Visit to explore v)
7. change u.color from gray to black (finished)
8. set time $=$ time $+ 1$ and $u.f =$ time

It is clear that the time counter increases by one every time we change the color of a vertex (from white to gray or from gray to black). We use $u.d$ and $u.f$ to record the two timestamps when DFS discovers and finishes with u, respectively. Before discussing properties of DFS, what is the total running time of DFS(G)? Its initilization of course costs $\Theta(n)$.

During the execution of DFS(G), we make exactly one call DFS-Visit(G, u) for each $u \in V$ because it is only in DFS-Visit (G, u) that we change the color of u from white to gray and from gray to black (and never touch it again). The running time of DFS-Visit(G, u), except those recursive calls made on line 6, is

$$\Theta(1) + \Theta\Big(|\mathrm{adj}(u)|\Big)$$

As a result, the total running time is

$$\Theta(n) + \sum_{u \in V} \Big(\Theta(1) + \Theta\Big(|\mathrm{adj}(u)|\Big)\Big) = \Theta(n + m)$$

where $n = |V|$ and $m = |E|$ are the number of vertices / edges.

Basic properties of DFS (G): (Prove them by yourself.)

1. At the time $u.d$ when a vertex $u \in V$ is discovered, the set of gray vertices is exactly the set of ancestors of u in the forest.
2. By the time $u.f$ when we finish with $u \in V$ (and thus, change its color u.color from gray to black), all vertices $v \in \mathrm{adj}(u)$ are either gray or black (meaning they have been discovered).

But why is DFS useful? To describe its first application, we need to prove the following key theorem: Let u, v be two vertices in G. We say u is an ancestor of v (or v is a descendant of u) if u, v lie in the same tree of the depth-first forest and u is an ancestor of v in the tree. We say u and v are unrelated if u is not an ancestor of v and v is not an ancestor of u (but may lie in the same tree).

Theorem (White-Path Theorem)

Given $u, v \in V$, u is an ancestor of v in the depth-first forest if and only if at the time $u.d$ (right before changing u from white to gray), there is a path from u to v consisting of white vertices.

The White-Path theorem is very powerful. For example, it implies that when we make a call DFS-Visit(G, u) in the for-loop of DFS (G), the tree rooted at u in the depth-first forest consists of exactly the vertices $v \in V$ such that there is a white path from u to v at the time $u.d$ (or equivalently, at the beginning of DFS-Visit (G, u)). To prove it, we need the following Parenthesis lemma:

Theorem

For any u and v, if u is an ancestor of v, then $[v.d, v.f]$ *is contained entirely within* $[u.d, u.f]$*:*

$$u.d < v.d < v.f < u.f$$

Similarly, if v is ancestor of u then $v.d < u.d < u.f < v.f$*.*
If u, v *are unrelated then the two intervals are entirely disjoint:*

$$u.f < v.d \quad \text{or} \quad v.f < u.d$$

We demonstrate this lemma with Figure 22.5. The proof is relatively straight-forward, using the two properties mentioned earlier, and can be found in the textbook.

We now use it to prove the White-Path theorem. We first show that if v is a descendant of u in the forest, then at time $u.d$ there is a white path from u to v. To this end, it suffices to show that at the time $u.d$, every vertex in the subtree rooted at u in the forest is white. (This clearly holds for u itself, check carefully the statement of the theorem.) For each proper descendant v of u, we have

$$u.d < v.d$$

by the Parenthesis lemma, so v is white at the time $u.d$.

The other direction is slightly more difficult: If at the time $u.d$, there is a white path from u to v, then v is a descendant of u. We assume for contradiction that $u, u_1, \ldots, u_k, v$ is a white path from u to v at the time $u.d$ for some $k \geq 0$, but v is not a descendant of u. Without loss of generality, we may assume that $u_1, \ldots, u_k$ are all descendants of u; otherwise we can choose u be the closest vertex to u along this path that is not a descendant of u. Denote u_k by w for convenience.

By the Parenthesis lemma, we have

$$w.f \leq u.f$$

(here we use $\leq$ instead of $<$ because w might well be u itself). Because v must be discovered before w is finished, we have

$$u.d < v.d < v.f < w.f \leq u.f$$

we conclude that v is a descendant of u, contradiction.

Now we describe the first application of DFS: Topological Sort. The input is a directed and acyclic graph (DAG) $G = (V, E)$. Here acyclic means that there is no cycle in G. We need to find a topological sort (i.e., a permutation) of the $n = |V|$ vertices such that for any edge $(u, v) \in E$, u appears before v in the sort. (It is clear that if G has a cycle then no topological sort exists.) For example, in task scheduling, the vertices are tasks and an edge $(u, v) \in E$ means that task u must be done before v. A topological sort of the vertices then gives a feasible order of the tasks, with no violation to the requirements from E.

Topological sort of a DAG $G = (V, E)$:

1. call DFS(G)
2. as each vertex is finished, insert it onto the front of a list
3. return the linked list of vertices

It is clear that the output is a permutation of V: $v_1, v_2, \dots, v_n$ sorted using their finishing timestamps:

$$v_1.f > v_2.f > \cdots > v_n.f$$

Running time of the algorithm is clearly $\Theta(n + m)$.

To prove the correctness of the algorithm, it suffices to show that given any DAG $G = (V, E)$, every edge $(u, v) \in E$ satisfies

$$u.f > v.f$$

This inspires us to classify, given the depth-first forest of DFS(G), each edge $(u, v) \in E$ into the following four types:

Given a directed graph $G = (V, E)$ (for now we do not require G to be acyclic) and its depth-first forest, we say $(u, v) \in E$ is a

1. Tree edge, if (u, v) is an edge in the forest (and $v.\pi = u$).
2. Back edge, if v is an ancestor of u in the forest.
3. Forward edge, if v is a descendant of u in the forest.
4. Cross edge, if u and v are unrelated (either they belong to different trees, or belong to the same tree but u is not an ancestor of v and v is not an ancestor of u).

Which type an edge $(u, v) \in E$ is depends on the color of v when DFS explores v in DFS-Visit(G, u) (as it goes through $\mathrm{adj}(u)$):

1. If v is white, then DFS will explore v and set $v.\pi = u$. So (u, v) is a tree edge and by the Parenthesis lemma: $u.f > v.f$.
2. If v is gray, then v is an ancestor of u and thus, (u, v) is a back edge. By the Parenthesis lemma, we have $v.f > u.f$.
3. If v is black, then (u, v) is either a forward edge or a cross edge. In both cases, we have $u.f > v.f$ (why?).

To summarize, for tree / forward / cross edges we have $u.f > v.f$, while for back edges we have $u.f < v.f$. (Keep this in mind because we will use it again in the next application of DFS: strongly connected components.) Now we prove the correctness of the Topological Sort algorithm. Let G be a DAG and we examine an edge $(u, v) \in E$. To show $u.f > v.f$, we follow the four cases. If (u, v) is a tree / forward / cross edge, we have already shown that $u.f > v.f$. The correctness follows if we can show that in a DAG, there is no back edge. This is trivial because a back edge implies a cycle (why?) and violates with the DAG assumption.