Depth First Search

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong

We have already learned breadth first search (BFS). Today, we will discuss its "sister version": the depth first search (DFS) algorithm. Our discussion will once again focus on directed graphs, because the extension to undirected graphs is straightforward.

DFS is a surprisingly powerful algorithm, and solves several classic problems elegantly. In this lecture, we will see one such problem—detecting whether the input graph contains cycles.

Paths and Cycles

Let $G = (V, E)$ be a directed graph.

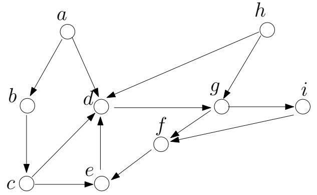
Recall:

A path in G is a sequence of edges $(v_1, v_2), (v_2, v_3), ..., (v_\ell, v_{\ell+1})$, for some integer $\ell \geq 1$. We may also denote the path as $v_1 \to v_2 \to ... \to v_{\ell+1}$.

We now define:

A path $v_1 \to v_2 \to ... \to v_{\ell+1}$ is called a cycle if $v_{\ell+1} = v_1$.

Example

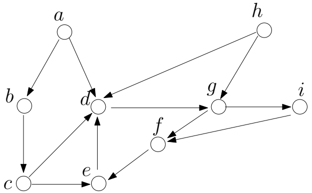


A cycle: $d \to g \to f \to e \to d$.
Another one: $d \to g \to i \to f \to e \to d$.

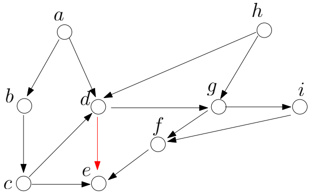
Directed Acyclic/Cyclic Graphs

If a directed graph contains no cycles, we say that it is a directed acyclic graph (DAG). Otherwise, G is cyclic.

Example



Cyclic



DAG

The Cycle Detection Problem

Let $G = (V, E)$ be a directed graph. Determine whether it is a DAG.

Next, we will describe the depth first search (DFS) algorithm to solve the problem in $O(|V| + |E|)$ time, which is optimal (because any algorithm must at least see every vertex and every edge once in the worst case).

Just like BFS, the DFS algorithm also outputs a tree, called the DFS-tree. This tree contains vital information about the input graph that allows us to decide whether the input graph is a DAG.

DFS

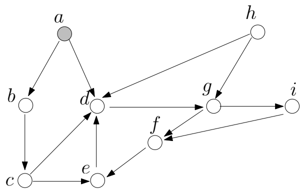
At the beginning, color all vertices in the graph white. And create an empty DFS tree T.

Create a stack S. Pick an arbitrary vertex v. Push v into S, and color it gray (which means "in the stack").

Make v the root of T.

Example

Suppose that we start from a.



DFS tree
a

$S = (a)$.

DFS

Repeat the following until S is empty.

1. Let v be the vertex that currently tops the stack S (do not remove v from S).
2. Does v still have a white out-neighbor?

   2.1 If yes: let it be u.

   - Push u into S, and color u gray.
   - Make u a child of v in the DFS-tree T.

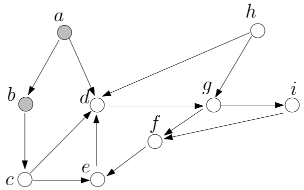
   2.2 If no, pop v from S, and color v black (meaning v is done).

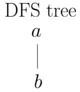
If there are still white vertices, repeat the above by restarting from an arbitrary white vertex v', creating a new DFS-tree rooted at v'.

DFS behaves like "exploring the web one click at a time", as we will see next.

Running Example

Top of stack: a, which has white out-neighbors b, d. Suppose we access b first. Push b into S.

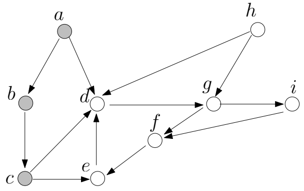


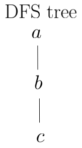


$S = (a, b)$.

Running Example

After pushing c into S:

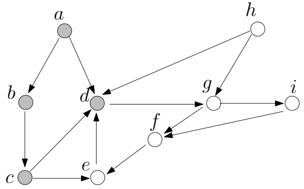




$S = (a, b, c)$.

Running Example

Now *c* tops the stack. It has white out-neighbors *d* and *e*. Suppose we visit *d* first. Push *d* into *S*.



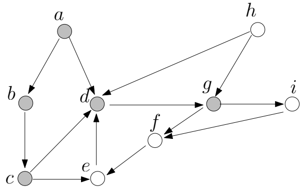
DFS tree

$$\begin{tikzcd}
a \arrow[d, no head] \\
b \arrow[d, no head] \\
c \arrow[d, no head] \\
d
\end{tikzcd}$$

$S = (a, b, c, d)$.

Running Example

After pushing g into S:



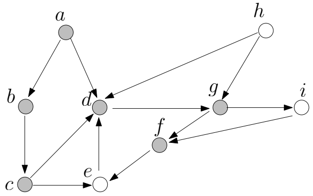
DFS tree

$$\begin{tikzcd} a \arrow[d, dash] \\ b \arrow[d, dash] \\ c \arrow[d, dash] \\ d \arrow[d, dash] \\ g \end{tikzcd}$$

$S = (a, b, c, d, g)$.

Running Example

Suppose we visit white out-neighbor f of g first. Push f into S



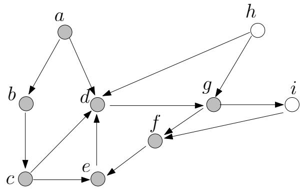
DFS tree

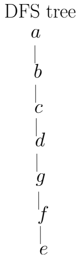
$$\begin{tikzcd}[row sep=small]
a \arrow[d, dash] \\
b \arrow[d, dash] \\
c \arrow[d, dash] \\
d \arrow[d, dash] \\
g \arrow[d, dash] \\
f
\end{tikzcd}$$

$S = (a, b, c, d, g, f)$.

Running Example

After pushing e into S:

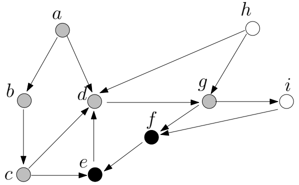


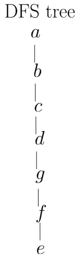


$S = (a, b, c, d, g, f, e)$.

Running Example

e has no white out-neighbors. So pop it from S, and color it black. Similarly, f has no white out-neighbors. Pop it from S, and color it black.

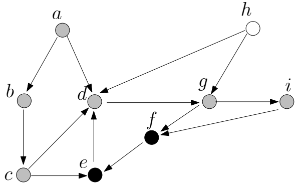


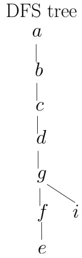


$S = (a, b, c, d, g)$.

Running Example

Now g tops the stack again. It still has a white out-neighbor i. So, push i into S.

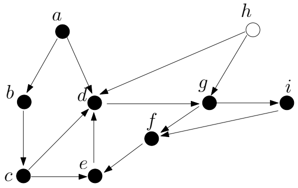




$S = (a, b, c, d, g, i)$.

Running Example

After popping i, g, d, c, b, a:



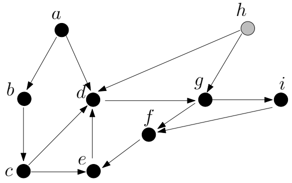
DFS tree

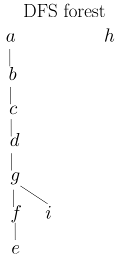
$$\begin{tikzcd}[row sep=small]
a \arrow[d, dash] \\
b \arrow[d, dash] \\
c \arrow[d, dash] \\
d \arrow[d, dash] \\
g \arrow[d, dash] \arrow[dr, dash] & \\
f \arrow[d, dash] & i \\
e
\end{tikzcd}$$

$S = ()$.

Running Example

Now there is still a white vertex *h*. So we perform another DFS starting from *h*.

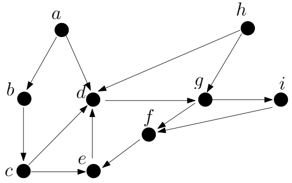


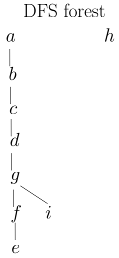


$S = (h)$.

Running Example

Pop *h*. The end.





$S = ()$.

Note that we have created a DFS-forest, which consists of 2 DFS-trees.

Time Analysis

DFS can be implemented efficiently as follows.

- Store G in the adjacency list format.
- For every vertex v, remember the out-neighbor to explore next.
- $O(|V| + |E|)$ stack operations.
- Use an array to remember the colors of all vertices.

Hence, the total running time is $O(|V| + |E|)$.

Recall that we said earlier that the DFS-tree (well, perhaps a DFS-forest) encodes valuable information about the input graph. Next, we will make this point specific, and solve the edge detection problem.

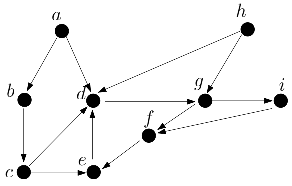
Edge Classification

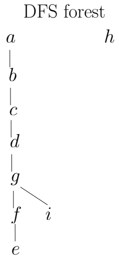
Suppose that we have already built a DFS-forest T.

Let (u, v) be an edge in G (remember that the edge is directed from u to v). It can be classified into

1. Forward edge: u is a proper ancestor of v in a DFS-tree of T.
2. Backward edge: u is a descendant of v in a DFS-tree of T.
3. Cross edge: If neither of the above applies.

Example





- Forward edges: $(a, b), (a, d), (b, c), (c, d), (c, e), (d, g), (g, f), (g, i), (f, e)$.
- Backward edge: (e, d).
- Cross edges: $(i, f), (h, d), (h, g)$.

After the DFS-forest T has been obtained, we can determine type of each edge (u, v) in constant time. All that is required is to augment the DFS algorithm slightly by remembering when each vertex enters and leaves the stack.

Augmenting DFS Slightly

Maintain a counter c, which is initially 0. Every time a push or pop is performed on the stack, we increment c by 1.

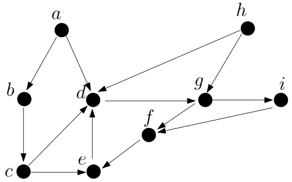
For every vertex v, define:

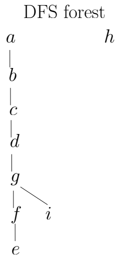
- Its discovery time $d\text{-}tm(v)$ to be the value of c right after v is pushed into the stack.
- Its finish time $f\text{-}tm(v)$ to be the value of c right after v is popped from the stack.

Define $I(v) = [d\text{-}tm(v), f\text{-}tm(v)]$.

It is straightforward to obtain $I(v)$ for all $v \in V$ by paying $O(|V|)$ extra time on top of DFS's running time. (Think: Why?)

Example





- $I(a) = [1, 16]$
- $I(b) = [2, 15]$
- $I(c) = [3, 14]$
- $I(d) = [4, 13]$
- $I(g) = [5, 12]$
- $I(f) = [6, 9]$
- $I(e) = [7, 8]$
- $I(i) = [10, 11]$
- $I(h) = [17, 18]$

Parenthesis Theorem

Theorem: All the following are true:

- If u is a proper ancestor of v in a DFS-tree of T, then $I(u)$ contains $I(v)$.
- If u is a proper descendant of v in a DFS-tree of T, then $I(u)$ is contained in $I(v)$.
- Otherwise, $I(u)$ and $I(v)$ are disjoint.

Proof: Follows directly from the first-in-last-out property of the stack. □

Cycle Theorem

Theorem: Let T be an arbitrary DFS-forest. G contains a cycle if and only if there is a backward edge with respect to T.

Proof: The "if-direction" is obvious. Proving the "only-if direction" is more involved, and will be done later. □

Cycle Detection

Equipped with the cycle theorem, we know that we can detect whether G has a cycle easily after having obtained a DFS-forest T:

- For every edge (u, v), determine whether it is a backward edge in $O(1)$ time.

If no backward edges are found, decide G to be a DAG; otherwise, G has at least a cycle.

Only $O(|E|)$ extra time is needed.

We now conclude that the cycle detection problem can be solved in $O(|V| + |E|)$ time.

It remains to prove the cycle theorem. We will first prove another important theorem, and then establish the cycle theorem as a corollary.

White Path Theorem

Theorem: Let u be a vertex in G. Consider the moment when u is pushed into the stack in the DFS algorithm. Then, a vertex v becomes a proper descendant of u in the DFS-forest if and only if the following is true:

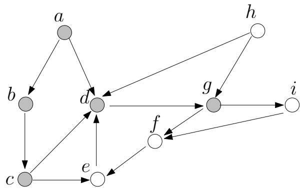
- We can go from u to v by traveling only on white vertices.

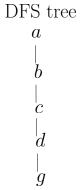
Proof: Will be left as an exercise (with solution provided). $\square$

Phrased differently, the theorem says that the search at u will get stuck only after discovering all the vertices that can still be discovered.

Example

Consider the moment in our previous example when g just entered the stack. $S = (a, b, c, d, g)$.





We can see that g can reach f, e, i by hopping on only white vertices. Therefore, f, e, i are all proper descendants of g in the DFS-forest; and g has no other descendants.

Proving the Only-If Direction of the Cycle Theorem

We will now prove that if G has a cycle, then there must be a backward edge in the DFS-forest.

Suppose that the cycle is $v_1 \to v_2 \to ... \to v_\ell \to v_1$.

Let v_i, for some $i \in [1, \ell]$, be the vertex in the cycle that is the first to enter the stack. Then, by the white path theorem, all the other vertices in the cycle must be proper descendants of v_i in the DFS-forest. This means that the edge pointing to v_i in the cycle is a back edge.

We thus have completed the whole proof of the cycle theorem. $\square$