Depth First Search

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong

Today, we will discuss the **depth first search** (DFS) algorithm, which is an elegant algorithm for solving many non-trivial problems. In this lecture, we will see one such problem: **cycle detection**.

The DFS algorithm has many interesting properties, among which the most important one is the **white path theorem**.

Paths and Cycles

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

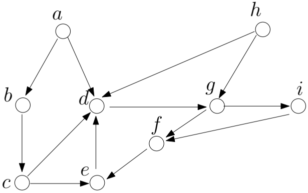
Recall:

> 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 $\ell \geq 1$ — where $v_1, v_2, ..., v_{\ell+1}$ are distinct vertices. We may also denote the path as $v_1 \to v_2 \to ... \to v_{\ell+1}$.

We now define:

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

Example

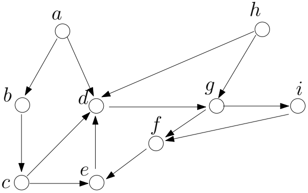


A cycle: $d \to g \to f \to e \to d$.
Another one: $d \to g \to i \to f \to e \to d$.

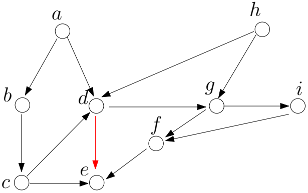
Directed Acyclic/Cyclic Graphs

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.

DFS outputs a tree, called the **DFS-tree**, which allows us to decide whether the input graph is a DAG.

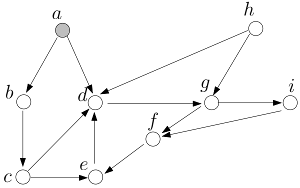
DFS

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?

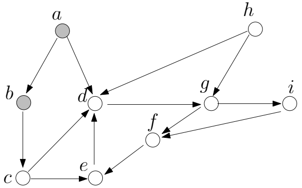
   2.1 If so, let it be u.
   - Push u into S, and color u **gray**.
   - Make u a child of v in the DFS-tree T.

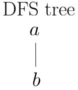
   2.2 Otherwise, pop v from S and color it **black** (meaning v is done).

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'.

Running Example

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

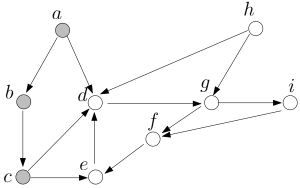


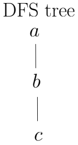


$S = (a, b)$.

Running Example

After pushing c into S:

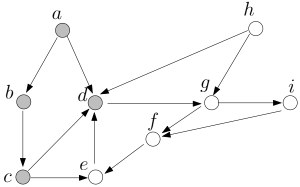


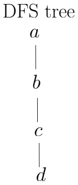


$S = (a, b, c)$.

Running Example

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

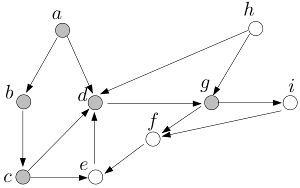


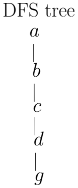


$S = (a, b, c, d)$.

Running Example

After pushing g into S:

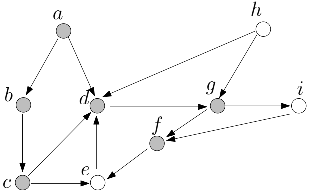


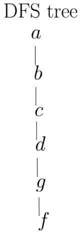


$S = (a, b, c, d, g)$.

Running Example

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

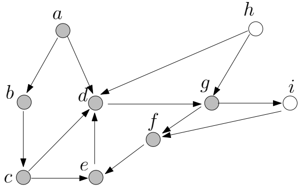


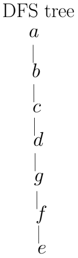


$S = (a, b, c, d, g, f)$.

Running Example

After pushing e into S:

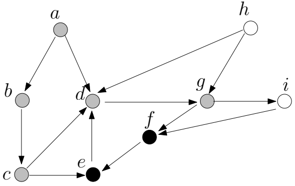


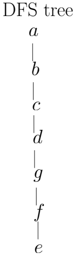


$S = (a, b, c, d, g, f, e)$.

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.

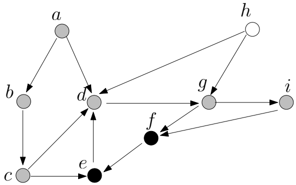


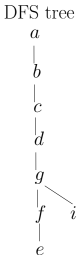


$S = (a, b, c, d, g)$.

Running Example

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

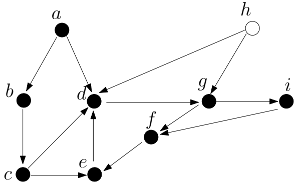


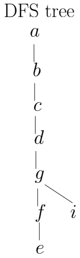


$S = (a, b, c, d, g, i)$.

Running Example

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

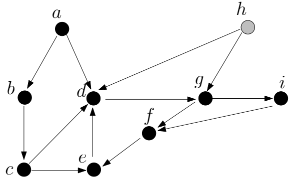


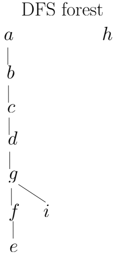


$S = ()$.

Running Example

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

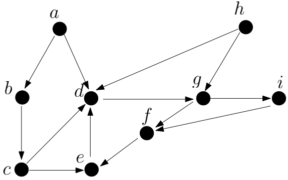


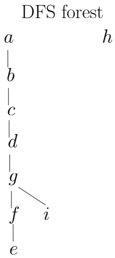


$S = (h)$.

Running Example

Pop h. The end.





$S = ()$.

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

The property below follows directly from the way DFS runs.

The Ancestor-Descendent Property: Let u and v be two distinct vertices in G. Then, u is an ancestor of v in the DFS-forest **if and only if** the following holds: u is already in the stack when v enters the stack.

Time Analysis

DFS can be implemented efficiently as follows.

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

The total running time is $O(|V| + |E|)$.

Next, we will see how to use the DFS forest to detect cycles.

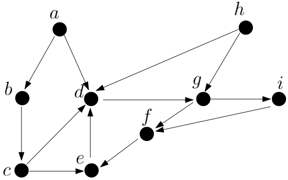
Edge Classification

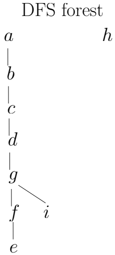
Suppose that we have already built a DFS-forest T.

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** if u is a proper ancestor of v in a DFS-tree of T;
2. **back edge** if u is a descendant of v in a DFS-tree of T;
3. **cross edge** if neither of the above applies.

Example





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

Cycle Theorem

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

The "if-direction" ($\Leftarrow$) is obvious. Proving the "only-if direction" ($\Rightarrow$) is more difficult and will be done later.

Issue: How to test the type of an edge?

We can do so in constant time. For this purpose, we need to slightly augment the DFS-forest by remembering when each vertex enters and leaves the stack.

Augmenting DFS

Maintain a counter c, which is initially 0. Every time we perform a push or pop, increment c by 1.

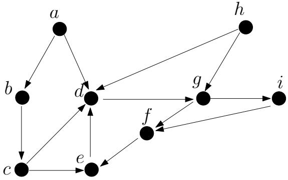
For every vertex v, define:

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

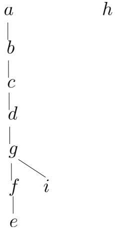
Define the **time interval** of v as $I(v) = [d\text{-}tm(v), f\text{-}tm(v)]$.

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



DFS forest



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

The property below follows directly from the stack's first-in-last-out property:

The Interval Property: For any two vertices u and v in G, their time intervals must satisfy one of the following:

- $I(u)$ contains $I(v)$;
- $I(v)$ contains $I(u)$;
- they are disjoint.

Combining the ancestor-descendant property with the interval property gives:

Theorem (the Parenthesis Theorem): Let u and v be two distinct vertices in G. Then:

- $I(u)$ contains $I(v)$ **if and only if** u is an ancestor of v in the DFS-forest.
- $I(u)$ and $I(v)$ are disjoint **if and only if** neither u nor v is an ancestor of the other.

Cycle Detection

We can now detect whether G has a cycle:

> **for** every edge (u, v) in G **do**
>   **if** $I(v)$ contains $I(u)$ **then**
>   **return** "cycle exists"
> **return** "no 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. In fact, it is a corollary of the **white path theorem**, another important theorem about DFS.

White Path Theorem

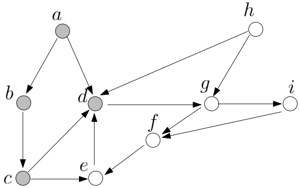
Theorem: Let u be a vertex in G. Consider the moment right before u enters the stack in the DFS algorithm. Then, a vertex v becomes a proper descendant of u in the DFS-forest **if and only** if the following is true at this moment:

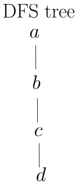
- there is a path from u to v including only white vertices.

We will prove the theorem at the end of this lecture.

Example

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





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

Proving the Only-If Direction ($\Rightarrow$) of the Cycle Theorem

We will now prove that if G has a cycle, then there must be a back 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. Hence, at the moment right before v_i enters the stack, v_i can reach all the other vertices in the cycle via white paths. By the white path theorem, all the other vertices in the cycle must be proper descendants of v_i in the DFS-forest. Hence, the edge pointing to v_i in the cycle must be a back edge. □

Proof of the White Path Theorem

Theorem: Let u be a vertex in G. Consider the moment right before u enters 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 at this moment:

- there is a path from u to v including only white vertices.

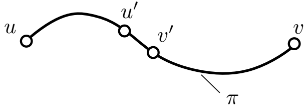
Proof: **The "only-if direction" ($\Rightarrow$):** Let

- v be a descendant of u in the DFS tree;
- π be the path from u to v in the tree.

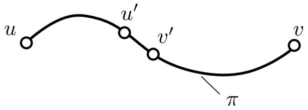
Clearly, π is also a path from u to v in G. All the nodes on π except for u are proper descendants of u and, by the ancestor-descendant property, must enter the stack after u. Hence, π must be white at the moment right before u enters the stack.

The "if direction" ($\Leftarrow$): Right before u enters the stack, a white path π exists from u to v. We will prove that all the vertices on π must be descendants of u in the DFS-forest.

Suppose that this is not true. Let v' be the first vertex on π — in the order from u to v — that is **not** a descendant of u in the DFS-forest. Clearly $v' \neq u$. Let u' be the vertex preceding v' on π (note: u' may be u).



By the way u' is defined, it must be a descendant of u in the DFS-forest.



Consider the moment when u' turns **black** (i.e., u' leaving the stack).

1. The color of v' cannot be white.

   Otherwise, v' is a white out-neighbor of u', in which case u' cannot be turning black.

2. Hence, the color of v' must be gray or black.

   Recall that when u entered stack, v' was white. Therefore, v' must have been pushed into the stack while u was still in the stack. By the ancestor-descendant property, v' must be a descendant of u in the DFS-forest. This, however, contradicts the definition of v'.

$\square$