Graph Decomposition*

Xiaofeng Gao

Department of Computer Science and Engineering
Shanghai Jiao Tong University, P.R.China

Algorithm Course @ Shanghai Jiao Tong University

*Special Thanks is given to Prof. Yijia Chen for sharing his teaching materials.

Exploring Graphs

Algorithm 1: $\textsc{Explore}(G, v)$

Input: $G = (V, E)$ is a graph; $v \in V$
Output: $\textsc{Visited}(u)$ = *true* for all nodes u **reachable** from v

```
VISITED(v) = true;
PREVISIT(v);
foreach edge (v, u) ∈ E do
    if not VISITED(u) then
        EXPLORE(G, u);
POSTVISIT(v);
```

- ▷ PREVISIT, POSTVISIT procedures are optional.
- ▷ work on a vertex when first discovered and left for the last time.

Correctness Proof

Theorem: EXPLORE(G, v) is correct (it visits all nodes reachable from v).

Proof: Every node it visits must be reachable from v:
EXPLORE moves from node to their neighbors; it can never jump to a region not reachable from v.

Every node reachable from v must be visited:
If $\exists u$ that EXPLORE misses, choose a path from v to u. Let z be the last vertex on that path that EXPLORE visited. Let w be the node immediately after it on this path.

So z was visited but w was not. This is a contradiction: while EXPLORE was at node z, it would have noticed w and moved on to it.

Depth-First Search

Algorithm 2: DFS(G)

Input: $G = (V, E)$ is a graph

Output: VISITED(v) is set to *true* for all nodes $v \in V$

foreach $v \in V$ **do**
    VISITED$(v) = false$;
foreach $v \in V$ **do**
    if *not* VISITED(v) **then**
        EXPLORE(G, v);

Running Time of DFS

Because of the VISITED array, each vertex is EXPLORE'd just *once*.

During the exploration of a vertex, there are the following steps:

- Some fixed amount of work – marking the spot as visited, and the PRE/POSTVISIT.

  The total work done in this step is then $O(|V|)$.

- A loop in which adjacent edges are scanned, to see if they lead somewhere new.

  Over the course of the entire DFS, each edge $(x, y) \in E$ is examined exactly *twice*, once during $\text{EXPLORE}(G, x)$ and once during $\text{EXPLORE}(G, y)$. The overall time is therefore $O(|E|)$.

Thus the depth-first search has a running time of $O(|V| + |E|)$.

Connectivity in Undirected Graphs

When EXPLORE starts at vertex v, it identifies the connected component containing v.

Each time the DFS outer loop calls EXPLORE, a new connected component is picked out $\Rightarrow$ can check if G is connected.

More generally, assign each node v an integer $\text{CCNUM}[v]$ to identify the connected component to which it belongs.

<u>PREVISIT</u>(v)

$\text{CCNUM}[v] = cc$

Initially, $cc = 0$, will increment each time DFS calls EXPLORE.

Previsit and postvisit orderings

For each node, we will note down the times of two important events:

- ▷ the moment of first discovery (corresponding to PREVISIT);
- ▷ and the moment of final departure (POSTVISIT).

<u>PREVISIT</u>(v)

$\text{PRE}[v] = \text{clock}$

$\text{clock} = \text{clock} + 1$

<u>POSTVISIT</u>(v)

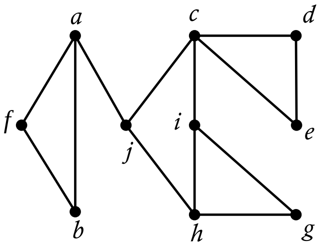
$\text{POST}[v] = \text{clock}$

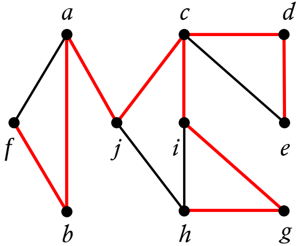
$\text{clock} = \text{clock} + 1$

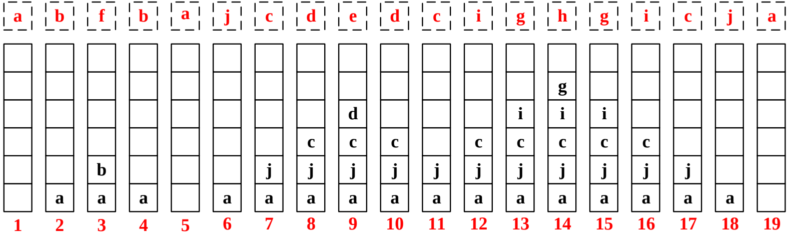
Lemma: $\forall u, v \in V$, intervals $[\text{PRE}(u), \text{POST}(u)]$, $[\text{PRE}(v), \text{POST}(v)]$ are either *disjoint* or *one is contained within the other*.

An executing example

Assume we use alphabetical order to explore G:







Types of Edges

DFS yields a **search tree/forests**: root; parent and child; descendant and ancestor.

- **Tree edges**: part of the DFS forest.
- **Forward edges**: lead from a node to a nonchild descendant in the DFS tree.
- **Backedges**: lead to an ancestor in the DFS tree.
- **Cross edges**: neither descendant nor ancestor; they lead to a node that has already been explored (that is, already postvisited).

PRE/POST ordering for (u, v)	Edge type
$[_u \quad [_v \quad]_v \quad]_u$	Tree/forward
$[_v \quad [_u \quad]_u \quad]_v$	Back
$[_v \quad]_v \quad [_u \quad]_u$	Cross

Directed Acyclic Graphs (DAG)

Lemma: A directed graph has a cycle if and only if its depth-first search reveals a back edge.

Proof: “$\Leftarrow$" (easy) If (u, v) is a back edge, then $\exists$ a cycle consisting of this edge together with the path from v to u in the search tree.

“$\Rightarrow$" Conversely, if the graph has a cycle $v_0 \rightarrow v_2 \rightarrow \cdots v_k \rightarrow v_0$, look at the first node v_i on this cycle to be discovered (the node with the lowest PRE number).

All the other v_j on the cycle are reachable from it and will therefore be its descendants in the search tree.

In particular, the edge $v_{i-1} \rightarrow v_i$ (or $v_k \rightarrow v_0$ if $i = 0$) is a back edge.

Linearization/Topologically Sort

Objective: Order the vertices such that every edge goes from a small vertex to a large one.

Lemma: In a dag, every edge leads to a vertex with a lower POST number.

Hence, a dag can be linearized by decreasing POST numbers, the vertex with the smallest POST number comes last in this linearization, and it must be a **sink** – no outgoing edges. Symmetrically, the one with the highest POST is a **source**, a node with no incoming edges.

Lemma: Every dag has at least one source and at least one sink.

The guaranteed existence of a source suggests an alternative approach to linearization:

① Find a source, output it, and delete it from the graph.

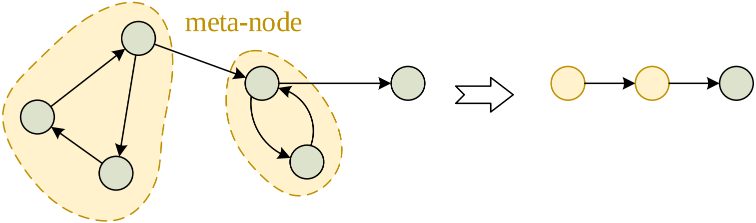
② Repeat until the graph is empty.

Connectivity for Directed Graphs

Definition: Two nodes u and v of a directed graph are **connected** if there is a path from u to v and a path from v to u.

This relation partitions V into disjoint sets that we call **strongly connected components**.

Lemma: Every directed graph is a dag of its strongly connected components.



Investigation

Lemma: If the EXPLORE subroutine is started at node u, then it will terminate precisely when all nodes reachable from u have been visited.

Therefore, if we call EXPLORE on a node that lies in a sink strongly connected component (a strongly connected component that is a sink in the meta-graph), then we will retrieve exactly that component.

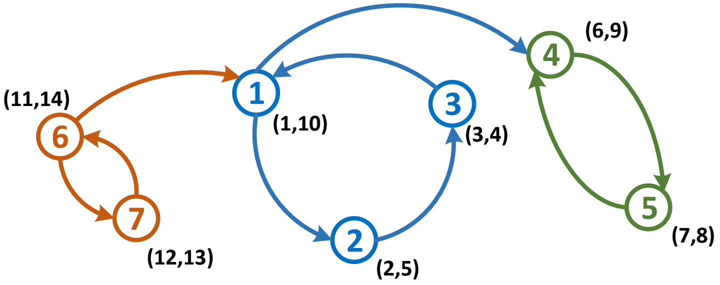
Lemma: If C and C' are strongly connected components, and there is an edge from a node in C to a node in C', then the highest POST number in C is bigger than the highest POST number in C'.

Lemma: The node that receives the highest POST number in a depth-first search must lie in a *source strongly connected component*.

Investigation (Cont')

Note: The smallest POST number in a depth-first search may NOT lie in a *sink strongly connected component*!

An Counter Example: (Node ID denotes the explore order)



The smallest POST number is Node 3, NOT in the sink strongly connected component (green).

An Efficient Algorithm

To design a linear-time algorithm, we have two problems:

(A) How do we find a node that we know for sure lies in a sink strongly connected component?

(B) How do we continue once this first component has been discovered?

Solving Problem A:

Consider the **reverse graph** G^R, the same as G but with all edges reversed (has exactly the same strongly connected components as G).

So, if we do a depth-first search of G^R, the node with the highest POST number will come from a source strongly connected component in G^R, which is a sink strongly connected component in G.

An Efficient Algorithm

Solving Problem B:

Once we have found the first strongly connected component and deleted it from the graph, the node with the highest POST number among those remaining will belong to a sink strongly connected component of whatever remains of G.

Thus we can keep using the post numbering from our initial depth-first search on G^R to successively output the second strongly connected component, the third strongly connected component, and so on.

The Linear-Time Algorithm:

1. Run depth-first search on G^R.
2. Run depth-first search on G, and process the vertices in decreasing order of their POST numbers from step 1.

Breadth-First Search

Algorithm 3: $\text{BFS}(G, s)$

Input: Graph $G = (V, E)$, directed or undirected; vertex $s \in V$

Output: $\text{DIST}(u)$ is set to the distance from s to all reachable u

```
foreach u ∈ V do
    DIST(u) = ∞;
DIST(s) = 0;
Q = [s] (queue containing just s);
while Q is not empty do
    u = EJECT(Q);
    foreach edge (u, v) ∈ E do
        if DIST(v) = ∞ then
            INJECT(Q, v);
            DIST(v) = DIST(u) + 1;
```

Correctness and efficiency

Lemma: For each $d = 0, 1, 2, \ldots$, there is a moment at which

(1) all nodes at distance $\leq d$ from s have their distances correctly set;

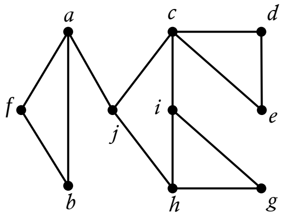
(2) all other nodes have their distances set to ∞; and

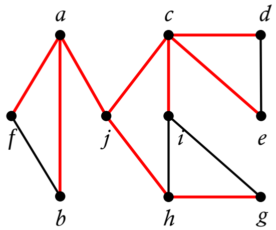
(3) the queue contains exactly the nodes at distance d.

Lemma: BFS has a running time of $O(|V| + |E|)$.

An executing example

Assume we use alphabetical order to explore G:





1		a					
2	a	b	f	j			
3	b	f	j				
4	f	j					
5	j	c	h				
6	c	h	d	e	i		
7	h	d	e	i	g		
8	d	e	i	g			
9	e	i	g				
10	i	g					
11	g						