Lecture 15: April 10, 2001

Today's Lecture: Graph Algorithms

- Depth-first search
- Topological sort
- Strongly connected components

Graph Searching Algorithms

- Systematic search of every edge and vertex of graph
- Graph $G = (V, E)$ is either directed or undirected
- Today's algs assume adjacency list representation

Examples:

- Depth-first search (DFS)
- Breadth-first search (BFS)

Applications:

- Compilers
- Graphics
- Maze-solving

DFS: Pseudocode

- input graph G may be undirected or directed.
- *time* is global variable used for time-stamping.

```
DFS(G)
for each vertex u ∈ V[G]
      do color[u] ← WHITE
time ← 0
for each vertex u ∈ V[G]
      do if color[u] = WHITE
            then DFS-VISIT(u)

DFS-VISIT(u)
color[u] ← GRAY                ▷ White vertex u discovered.
d[u] ← time                    ▷ Mark with discovery time.
time ← time + 1                ▷ Tick.
for each v ∈ Adj[u]            ▷ Explore all edges (u, v).
      do if color[v] = WHITE
            then DFS-VISIT(v)
color[u] ← BLACK               ▷ Blacken u; it is finished.
f[u] ← time                    ▷ Mark with finishing time.
time ← time + 1                ▷ Tick.
```


DFS: How it works

- Initialize all vertices to white
- Reset global counter
- Check each vertex; visit each WHITE vertex using DFS-VISIT
- Each call to DFS-VISIT(u) roots a new tree of depth-first forest at vertex v
- Vertex is GRAY if it has been discovered, but not all its edges have been explored!
- GRAY edges always form a linear chain!
- Vertex is BLACK after all its edges are explored
- When DFS returns, every vertex u assigned: a discovery time $d[u]$, and a finishing time $f[u]$

DFS: Running time

Running time $O(V^2)$, because
  DFS-VISIT called once per vertex
  Each loop over Adj runs $< |V|$ times.
But... can we show a better bound?

DFS: running time

- **Amortized bookkeeping:** charge exploration of edge *to* the edge:

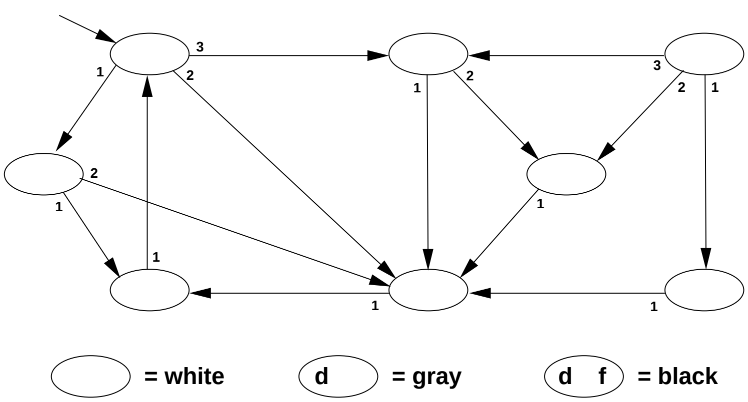
  Charge DFS-VISIT loop body to edge (runs once per edge if directed graph, twice if undirected)

  Charge rest of DFS-VISIT to vertex (runs once per vertex)

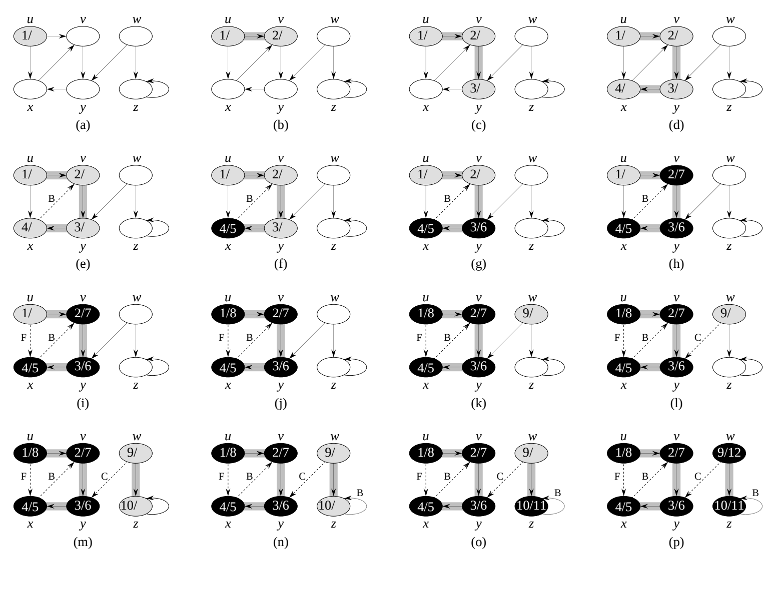
- Time $= O(V + E)$ – *linear time*

  $O(V + E)$ *is considered linear time for graph because it is linear in size of adjacency-list representation!*

DFS Example



DFS Example



DFS Timestamping

The procedure DFS records:

- discovery time of vertex u in $d[u]$
- finishing time of vertex u in $f[u]$

For every vertex u,
$d[u] < f[u]$.

DFS: Structure of colored vertices

Vertex u is:

- WHITE before time $d[u]$
- GRAY between time $d[u]$ and time $f[u]$
- BLACK thereafter.

Also notice structure throughout algorithm:

- GRAY vertices form a linear chain.
  - stack of recursive calls
    (things started but not yet finished)

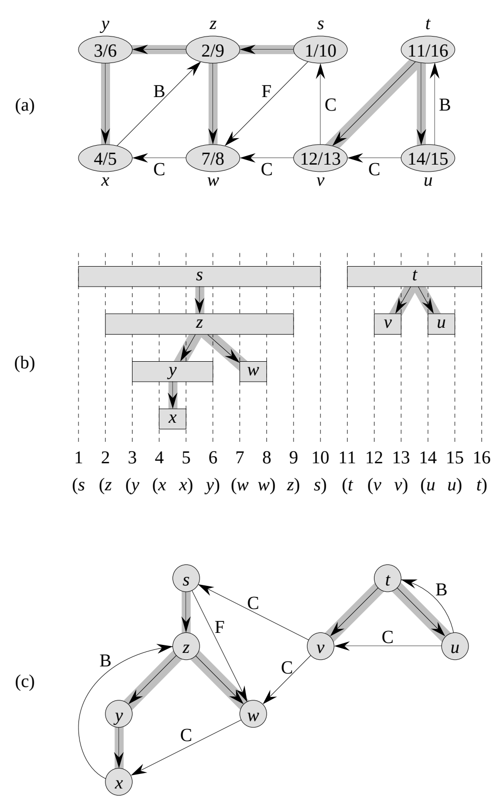
DFS: parenthesis theorem

Discovery, finish times have
parenthesis structure.

- represent discovery of u with left parenthesis "(u"
- represent finishing u by right parenthesis "u)"
- history of discoveries and finishings makes a well-formed expression! (Parentheses are properly nested.)

Proof in CLR (omitted here); intuition:
Intervals either disjoint or
enclosed, but never (otherwise) overlap
We'll just look at example.

DFS and Parenthesization



Edge Classification

Tree edge: (GRAY to WHITE)
encounter new (WHITE) vertex
Form spanning forest (no cycles)

Back edge: (GRAY to GRAY)
from descendant to ancestor

Forward edge: (GRAY to BLACK)
nontree, from ancestor to descendant

Cross edge: (GRAY to BLACK)
remainder — between trees or subtrees
(if same tree, can't go anc/desc, or desc/anc)

DFS: edge classification

Notes:

- ancestor/descendant is with respect to **tree** edges
- **tree** and **back** edges are important;
- most algorithms don't distinguish between **forward** and **cross** edges

Exercise:

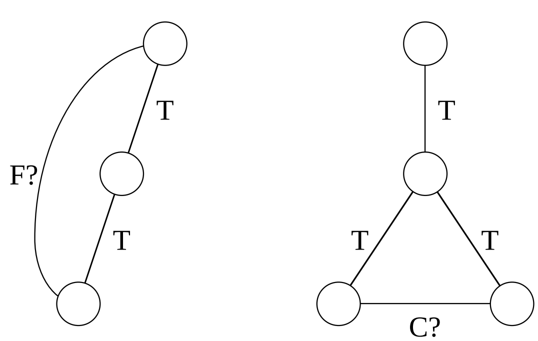
- How to distinguish forward, cross edges in DFS? (Hint: look at discovery times.)

DFS: Lemma

Lemma: (Theorem 23.9)

In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.

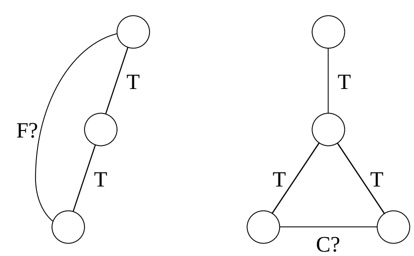
Sketch of proof:



DFS: Lemma

Lemma: (Theorem 23.9)

Proof:

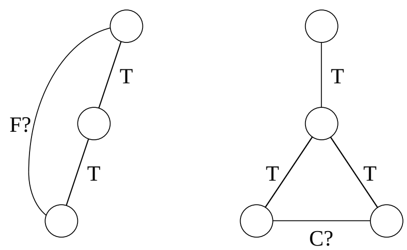


▷ Suppose there's a forward edge F? (at left)
But F? edge must actually be B because must finish processing bottom vertex before resuming with top vertex.

DFS: Lemma

Lemma: (Theorem 23.9)

Proof:



▹ Suppose there's a cross edge C? between subtrees (at right)

C? edge can't be Cross edge:
Must be explored from vertex it connects, becoming T, before other vertex is explored; so, two bottom T labels can't both be right — one must be a B.

Exercise

Can use DFS to find cycles!

An undirected graph is acyclic (i.e., a forest) iff a DFS yields no back edges.

- Acyclic $\Rightarrow$ no back edge:
  trivial (back edge $\Rightarrow$ cycle)
- No back edges $\Rightarrow$ acyclic:
  No back edges $\Rightarrow$ only tree edges (by above lemma) $\Rightarrow$ forest $\Rightarrow$ acyclic

Exercise

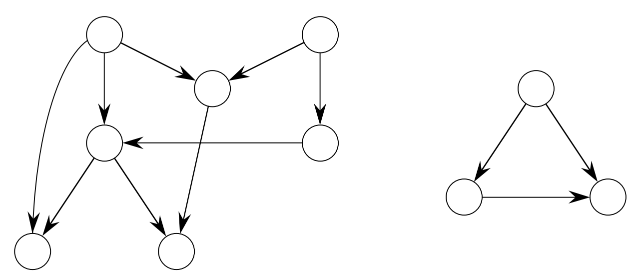
Thus can run DFS: if find a back edge, there's a cycle

- Time $O(V)$, [not $O(V+E)$!]

If ever see $|V|$ distinct edges, must have seen a back edge, because in acyclic (undirected) forest, $|E| \leq |V| - 1$.

Directed Acyclic Graphs (DAG)

- No *directed* cycles
  example:



- Used in many applications to indicate precedences among events
- Example: parallel code execution
  - Topological Sort (induce a total ordering)

DAG: Theorem

Theorem: A directed graph G is acyclic iff a DFS yields no back edges.
$\Rightarrow$: back edge $\Rightarrow$ cycle

$\Leftarrow$: Contrapositive: cycle $\Rightarrow$ back edge

Suppose G has a cycle. Let v have lowest discovery # on cycle, and let u be predecessor on cycle.

$$\begin{array}{ccc} u & \longrightarrow & v \\ \nwarrow & \cdots & \swarrow \end{array}$$

(v is first vertex visited)

When v discovered, whole cycle is WHITE.

Must visit everything reachable on a WHITE path from v before returning from DFS-VISIT(v).
Thus (u, v) is a back edge. $\square$

- $O(V + E)$ time [Why not $O(v)$ as before?]

Topological Sort

Topological Sort of a dag $G = (V, E)$ is:

- Linear ordering of all vertices of a dag

such that

- If G contains an edge (u, v), then u appears before v in the ordering.

If the graph has a cycle, then no linear ordering is possible!

Topological Sort: pseudocode

The following algorithm topologically sorts a DAG:

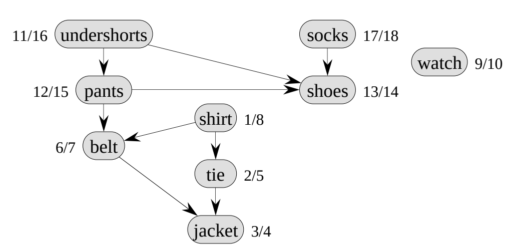
TOPOLOGICAL-SORT(G)
call DFS(G) to compute finishing times $f[v]$ for each vertex v
as each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices

At end, linked list comprises total ordering!

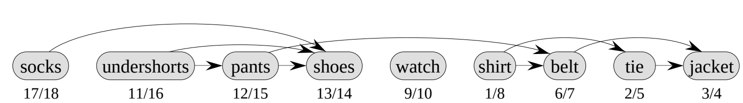
Topological Sort: Example

Example: precedence relations (don x before y)
Intuition: Can "schedule" task only when all of its subtasks have been scheduled

(a)



(b)



Topological Sort: running time

Running Time:

- depth-first search: takes $O(V+E)$ time
- insert each of the $|V|$ vertices onto the front of the linked list: takes $O(1)$

We can perform a topological sort in time $O(V+E)$.

Topological Sort: correctness

Correctness proof for TOPOLOGICAL-SORT(G)

Claim: $(u,v) \in E \Rightarrow f[u] > f[v]$

When (u,v) explored, u is GRAY

$v =$GRAY
$\Rightarrow (u,v) =$ backedge (cycle, contradiction).

$v =$WHITE
$\Rightarrow v$ becomes descendant of u
$\Rightarrow f[v] < f[u]$

$v =$BLACK
$\Rightarrow f[v] < f[u]$

Strongly Connected Components (SCC)

Definition:

A strongly connected component of a directed graph $G=(V,E)$ is:

a maximal set of vertices $U \subseteq V$ such that for every pair of vertices u and v in U, we have both

- $u \leadsto v$
  and
- $v \leadsto u$

That is, u and v are reachable from each other!

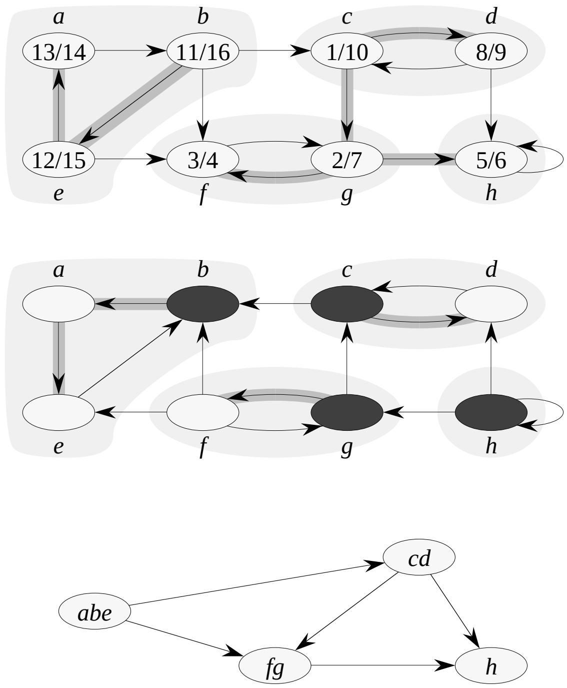
Strongly Connected Components

in other words ...

- u $\mathbf{R}$ v if u and v lie on a common cycle.
- $\mathbf{R}$ is an equivalence relation (r,s,t).
- strongly connected components are a partition of graph G under $\mathbf{R}$.

SCC: examples

(a)
(b)
(c)



SCC: Pseudocode

(CLR §23.5)

To compute SCC of directed graph $G = (V, E)$, use two DFS's, one on G and one on G^{T} (G, with edges swapped):

STRONGLY-CONNECTED-COMPONENTS(G)
1 call $\mathrm{DFS}(G)$ to compute finishing times $f[u]$ for each vertex u
2 compute G^{T}
3 call $\mathrm{DFS}(G^{\mathrm{T}})$, but in the main loop of DFS, consider the vertices in order of decreasing $f[u]$ (as computed in line 1)
4 output vertices of each tree in the depth-first forest of step 3 as a separate SCC

Intuition: explore latest-finished vertices first
Running time $\Theta(V + E)$ [Why?]

- Strongly-Connected-Components can be found in linear time.