Representing graphs

$G = (V, E)$

V = set of vertices

$E \subseteq V \times V$ = set of edges

$|E| = O(V^2)$

Note: drop $|\ |$ inside asymp. notation.

If G is connected, i.e., $\exists$ path $u \rightsquigarrow v$ $\forall u, v \in V$

then $|E| \geq |V| - 1$

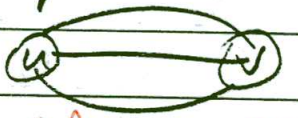
$\lg |E| = \Theta(\lg V)$

Ω from $|E| \geq |V| - 1$

O from $|E| = O(V^2) \Rightarrow \lg |E| = O(2 \lg V)$

Variations:

- undirected: edge $(u,v) = (v,u)$; no self-loops (by convention)
- directed: (u,v) is edge $u \rightarrow v$; (by convention, self-loops allowed)
- weighted: put weight on vertices or edges
- multigraph: allow multiple edges (u,v)



- hypergraph - "edges" can connect more than 2 vertices

e.g. all students
- vertices
all friendships & enmities
- hyperedges

Dense graph: $|E| \approx |V|^2$

Sparse graph: $|E| << |V|^2$

typically $|E| \approx |V|$

typically, sparse: $|E| = o(V)$

dense: $|E| = \Omega(V^2)$ or $\Omega(V^{1+\varepsilon})$ for some constant $\varepsilon > 0$

Assume $V = \{1, 2, \ldots, n\}$

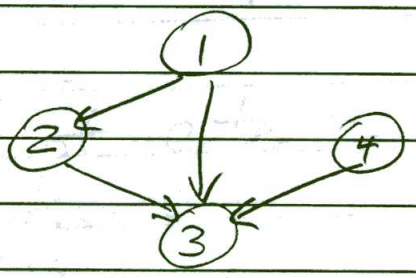
Adjacency matrix

A is $n \times n$

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{if } (i,j) \notin E. \end{cases}$$

If weighted,

$$a_{ij} = \begin{cases} w(i,j) & \text{if } (i,j) \in E, \\ 0 & \text{o.w.} \end{cases}$$



A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

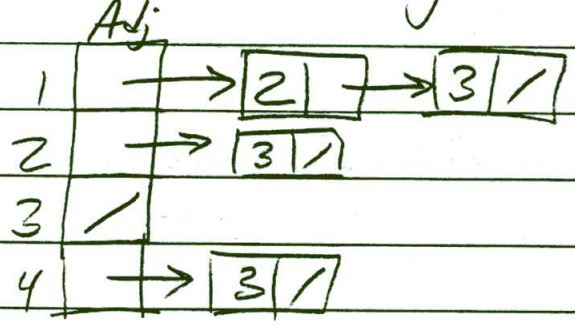
$\Theta(V^2)$ space $\Rightarrow$ OK for dense graphs, but wasteful for sparse graphs.

If G is undirected $\Rightarrow$

$a_{ij} = a_{ji} \Rightarrow A$ is symmetric

Adjacency lists

For each vertex v, keep a list $Adj[v]$ of vertices adjacent to v.



Could also keep list of incoming edges.

Storage:

degree of vertex = # of incident edges

out-degree = " " outgoing "

in-degree = " " incoming "

Directed graph: $\sum_{v \in V} \text{out-degree}(v) = |E|$

$\Rightarrow \Theta(V+E)$ storage

Undirected graph: $\sum_{v \in V} \text{degree}(v) = 2|E|$

(Each edge counted once for each vertex it's incident on.)

$\Rightarrow$ Also $\Theta(V+E)$ storage.

Which rep. is best depends on the application.

CS 25- Algorithms

11/3/95

Last time (chap 22, 23)
- DS- UF
- Graph Representation

Today (chap 23)
- Graph search

Handouts
- HW5- Graded
- HW6

Announcements
- Tues x-hour
- ~~Exams~~

CS25-X94 Lecture 17 8/3/94

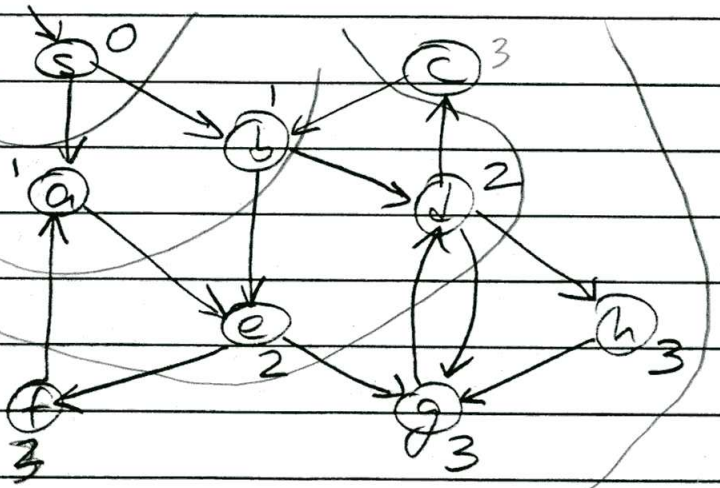
Check: hold x-hour next Tuesday to go over midterm solutions?

Today: searching a graph.
Omitting some details from CLR:
- predecessor subgraphs
- BFS correctness (it's special case of Dijkstra)

BFS

Input: $G = (V, E)$, adjacency list rep.
source vertex $s \in V$

Output: Finds all vertices reachable from s
$= \{v : s \rightsquigarrow v\}$
Also computes $d[v] =$ min # of edges on $s \rightsquigarrow v$ path. $d[v] = \infty$ if $s \not\rightsquigarrow v$.



Q = s
Q = ba
Q = eb

Idea: Expand a frontier.
Propagate a wave, one layer at a time
Use a FIFO queue to keep track of where to expand from.

```
BFS(V, E, s)
for each vertex v ∈ V − {s}                    ⎫
    do d[v] ← ∞                                ⎬ Θ(V)
d[s] ← 0                                       ⎭
Q ← empty queue
Enqueue(Q, s)
while Q not empty                              ⎫
    do u ← Dequeue(Q)                          ⎪
        for each vertex v ∈ Adj[u]             ⎬ O(VE)
            do if d[v] = ∞                     ⎪
                then d[v] ← d[u] + 1           ⎪
                     Enqueue(Q, v)             ⎭
```

once for each edge in graph

Note:
- Not computing predecessor subgraph.
- Not coloring vertices. CLR checks whether color[v] = WHITE, but it's sufficient to check whether d[v] = ∞.

Run on examples, showing d and Q.

Correctness: special case of Dijkstra's alg, which we'll see next week.

Time: $O(V+E)$.
- Each vertex enqueued at most once.
- $\Rightarrow$ each adj list scanned at most once.
- $\Rightarrow O(E)$
- $O(V)$ from initialization of d.
- Linear time, since adj list rep is $O(V+E)$

Note: May not reach all vertices.
Only reaches those for which $d[v]$ is set to $< \infty$.

DFS

Input: Same as for BFS.

Output: Search of entire graph.
$d[v]$ = discovery time of v
$f[v]$ = finish time of v
Classification of edges

- erase board
- start example first, then alg

```
DFS(V, E)
for each vertex u ∈ V
    do color[u] ← white
time ← 0          ▷ global
for each vertex u ∈ V
    do if color[u] = white
        then DFS-Visit(u)

DFS-Visit(u)
color[u] ← gray   ▷ discover white vertex u
time++
d[u] ← time
for each v ∈ Adj[u]   ▷ explore edge (u,v)
    do if color[v] = white
        then DFS-Visit(v)
color[u] ← black  ▷ finished exploring from u
time++
f[u] ← time
```

Colors:

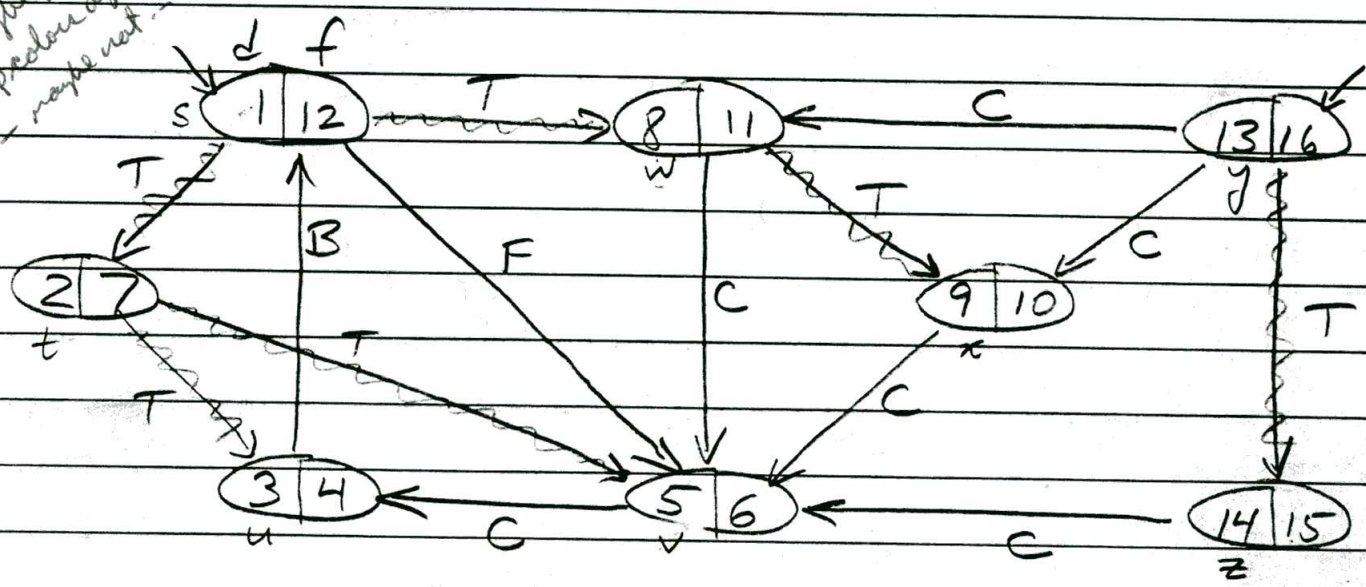
- white ⇒ not yet discovered
- gray ⇒ discovered, still exploring from it
- black ⇒ all done exploring from it

Time: $\Theta(V+E)$

- $\Theta(V)$ to initialize colors. Also, each vertex will be searched from, eventually.
- Each vertex discovered only once.
- $\Rightarrow$ search adj list of each vertex once
- $\Rightarrow \Theta(E)$ search time

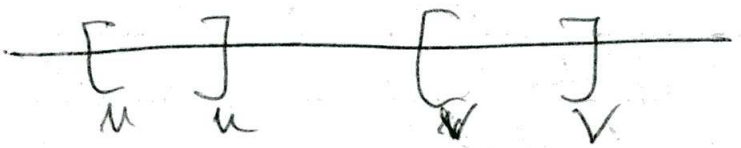
Example: (did example last time, but put up again to explain next couple of topics)

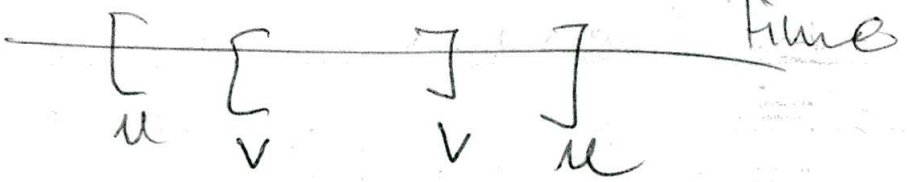
- maybe put procedure again?
- maybe not.



* V is gray between d[v] and f[v]
What is structure of gray vertices?
- Form stack of recursive calls of DFS-Visit.

Note: colors not really necessary — only need to know whether vertex has been discovered. But colors helps in reasoning.





NO:

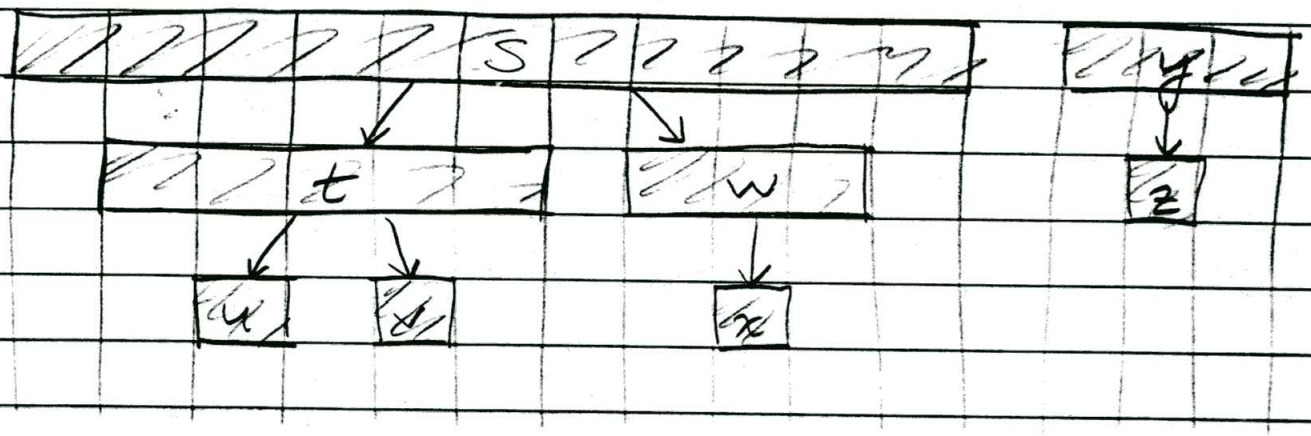
parantheses: finish the last one open

Parenthesis theorem:

After DFS, $\forall u, v \in V$, one of the following hold

1. intervals $[d[u], f[u]]$, $[d[v], f[v]]$ are disjoint
2. $[d[u], f[u]]$ contained in $[d[v], f[v]]$ and v is ancestor of u in DF tree.
3. $[d[v], f[v]]$ contained in $[d[u], f[u]]$ and u is ancestor of v.

For example above:



1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

(s (t (u u) (v v) t) (w (x x) w) s) (y (z z) y)

Proof: Since discovery times are unique either $d[u] < d[v]$ or $d[v] < d[u]$. Cases are symmetric, so assume w.l.o.g. that $d[u] < d[v]$.

① If $d[v] < f[u]$, then v discovered while u still gray $\Rightarrow u$ is ancestor of v. v discovered after $u \Rightarrow v$ finishes before $u \Rightarrow f[v] < f[u]$.

② If $f[u] < d[v]$, then $[d[u], f[u]]$, $[d[v], f[v]]$ are disjoint.

③

□

Cor: u is a proper ancestor of v iff
$d[u] < d[v] < f[v] < f[u]$. ☒

White-path theorem:
u is an ancestor of v iff at time $d[u]$,
$\exists$ path $u \rightsquigarrow v$ consisting only of white vertices.
Proof: Sort of obvious, but take a look. ☒

Classification of edges

- Tree: $a \to b$ (gray → white)

  Form a ~~spanning~~ forest. (No cycles, because tree edges are written when we see a white vertex.)

- Back: $a \to b$ (gray → gray)

  Descendant to ancestor

- Forward: $a \to b$ (gray → black), $d[a] < d[b]$

  Ancestor to descendant, but not tree.

- Cross: $a \to b$ (gray → black), $d[b] < d[a]$

  Between vertices in a tree (but neither is ancestor of other), or bet. trees.

CS 25 - Algorithms

11/6/95

Last time (chap 23)

- Graph search
  BFS, DFS

Today (chap 23)

- Finish DFS
- Topological sort
- Strongly connected components

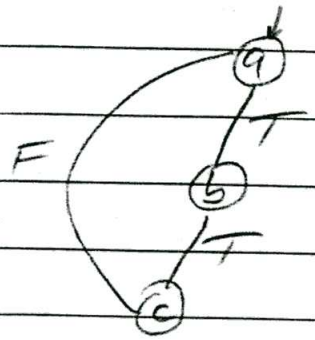
Announcement

- X-hour
  - return exams

CS25-X94 Lecture R 8/5/94

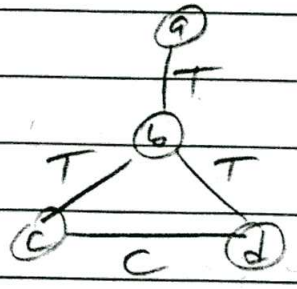
Theorem: If G is undirected, DFS produces only tree & back edges.

Proof: Assume ∃ forward edge:



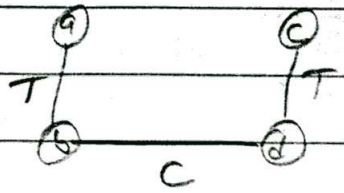
Edge (c,a) must be back, since finish processing c before resuming search from a.

Assume ∃ cross edge ~~both subtrees~~ within subtree:



(c,d) must be explored from either c or d, becoming T, before the other vertex is explored. In fact (b,c) and (b,d) can't both be T — one must be B.

Assume ∃ cross edge bet. trees:



(b,d) must be explored from either b or d, becoming T. ☒

discuss • converse • contrapositive

Theorem: acyclic ⇔ no back edge

An undirected graph is acyclic iff DFS yields no back edge.

Proof: ∃ back edge ⇒ cycle. (contrapositive of ⇒)

∄ back edge ⇒ only tree edges (since ∃ forward or cross ⇒ forest ⇒ acyclic. ⇐) ☒

Algorithm to determine whether undirected graph has a cycle:

Run DFS, stopping if find a back edge.

Time = $O(V)$, not $O(V+E)$.

Why? If ∃ back edge, you'll find it after at most $|V|$ edges.

Why? In an acyclic graph, $|E| \leq |V|-1$.

connected, acyclic graph $|E| = |V|-1$?

cyclic 

Define: DAGs Directed Acyclic Graph

Theorem: A directed graph G is acyclic iff a DFS of G yields no back edges.

Proof: (contrapositive) ⇒: ∃ back edge (u,v) ⇒ v is ancestor of u ⇒ ∃ path v⇝u ⇒ cycle v⇝u→v.

(contrapositive) ⇐: G contains a cycle c ⇒ let v = 1st vertex discovered on c & let (u,v) be preceding edge in c ⇒ at time d[v], ∃ path of white vertices v⇝u ⇒ by white-path thm, v is ancestor of u ⇒ (u,v) is a back edge. ☒

see book

- v in gray once discovered
- v still gray when u discovered
- u→v is gray→gray ⇒ back edge

∴ Can determine whether a directed graph is acyclic in O(V+E) time.

Topological sort

eg precedence ordering of events for construction project — foundation, framing, plumbing & electrical, drywall, septic, painting

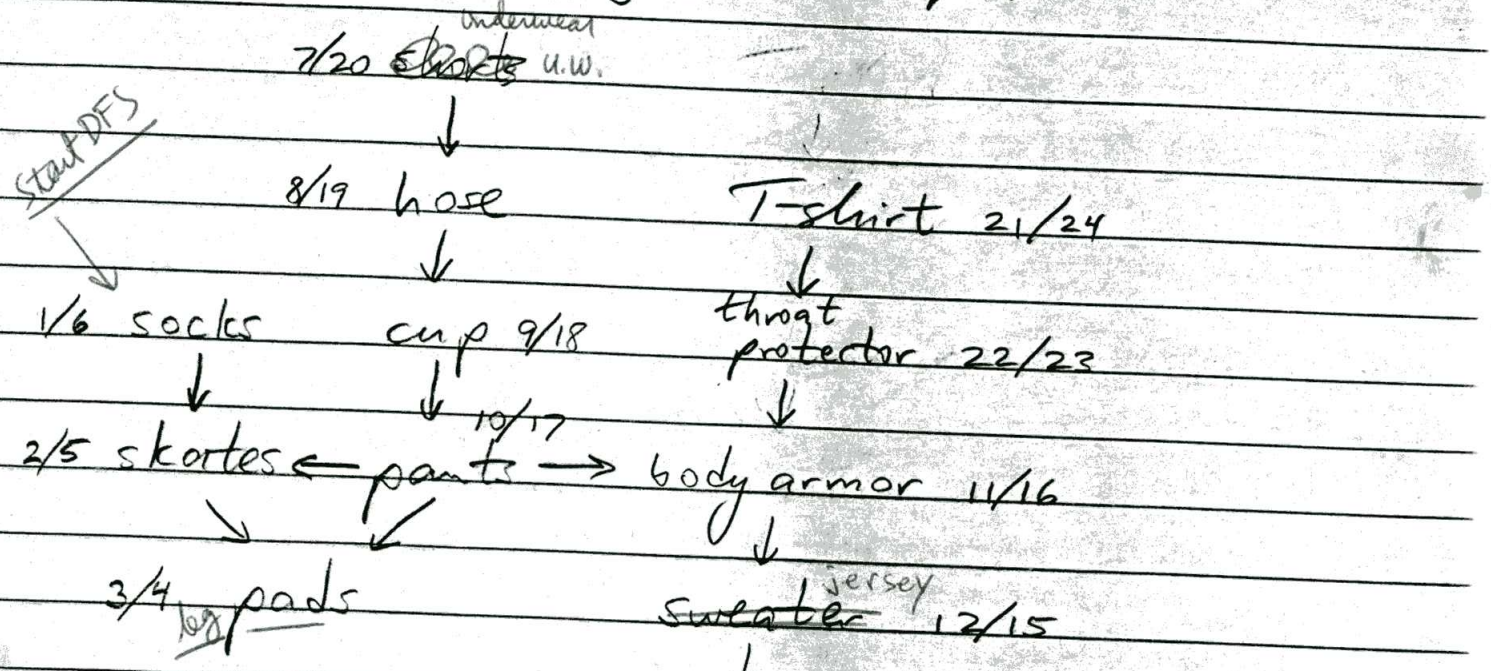
Topo sort of DAG G is a linear ordering of its vertices s.t. if G contains edge (u,v), then u appears before v.

Order vertices along a line so that all directed edges go L→R.

Not like the usual notion of sorting.

Application: construction project

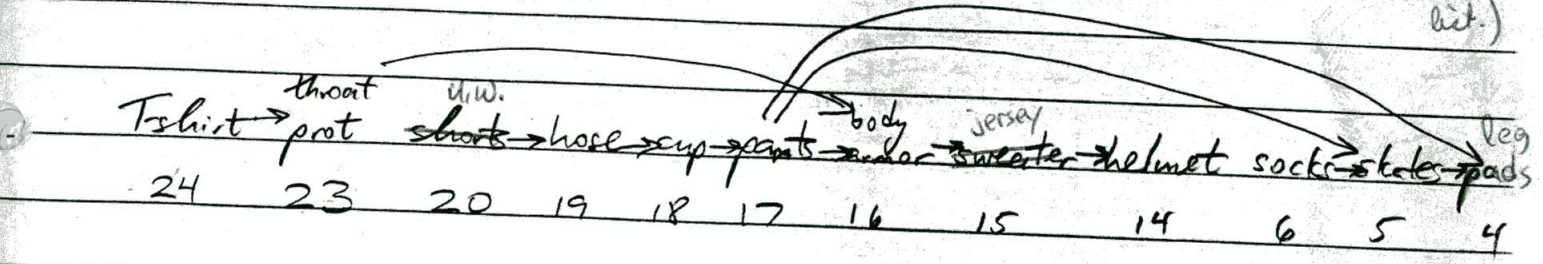
Application: donning goalie equipment



Topo-Sort(V,E)

- run DFS(V,E), ~~but when a vertex is finished, output it~~
  ~~reverse output order of vertices~~
- output vertices in reverse order of finish times (e.g., when vertex finished, insert it at the head of a linked list; return linked list.)

Above example:



Time = $O(V+E)$

Correctness: Only need to show that if $\exists$ edge (u,v), then ~~$f[v] < f[u]$~~. ($f[u] > f[v]$)

When (u,v) is explored, u is gray.

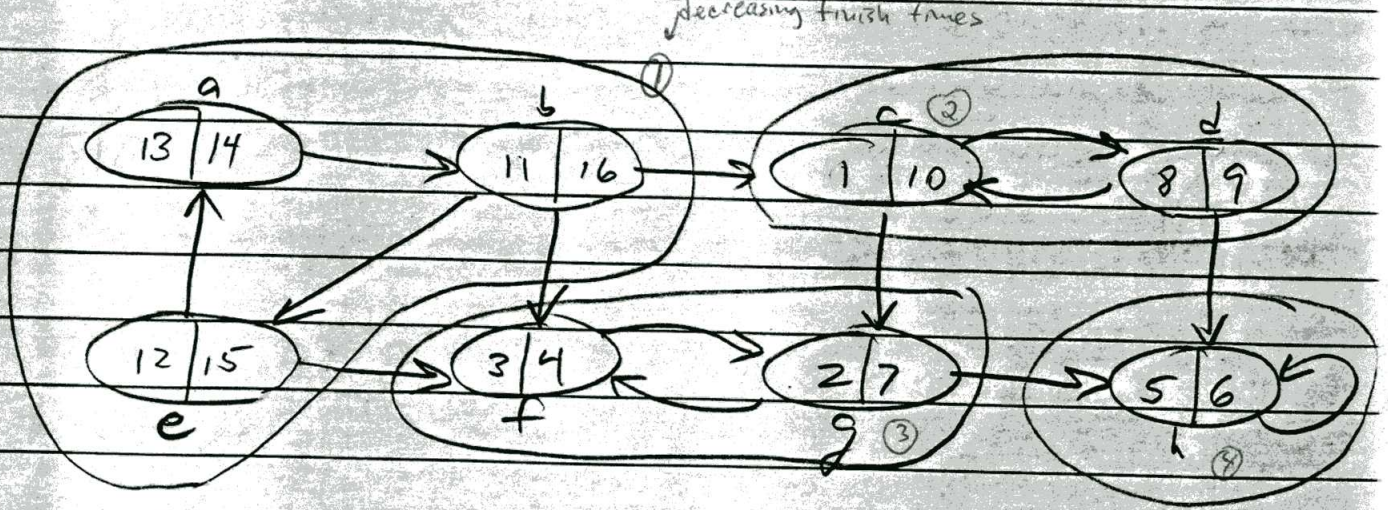
Consider color of v:

- v is gray $\Rightarrow$ (u,v) is a back edge $\Rightarrow$ contradiction (DAG has no back edges).
- v is white $\Rightarrow$ u is ancestor of v in depth-first tree $\Rightarrow$ ~~$f[v] < f[u]$~~. ($f[u] > f[v]$)
- v is black $\Rightarrow$ ~~$f[v] < f[u]$~~. ($f[u] > f[v]$)

v, done already, u still being processed.

Strongly connected components

SCC of directed $G=(V,E)$ is a maximal set of vertices $U \subseteq V$ s.t. $\forall u,v \in U$, $u \rightsquigarrow v$ and $v \rightsquigarrow u$. (u, v reachable from each other.)



Need transpose of G:

$G^T = (V, E^T)$

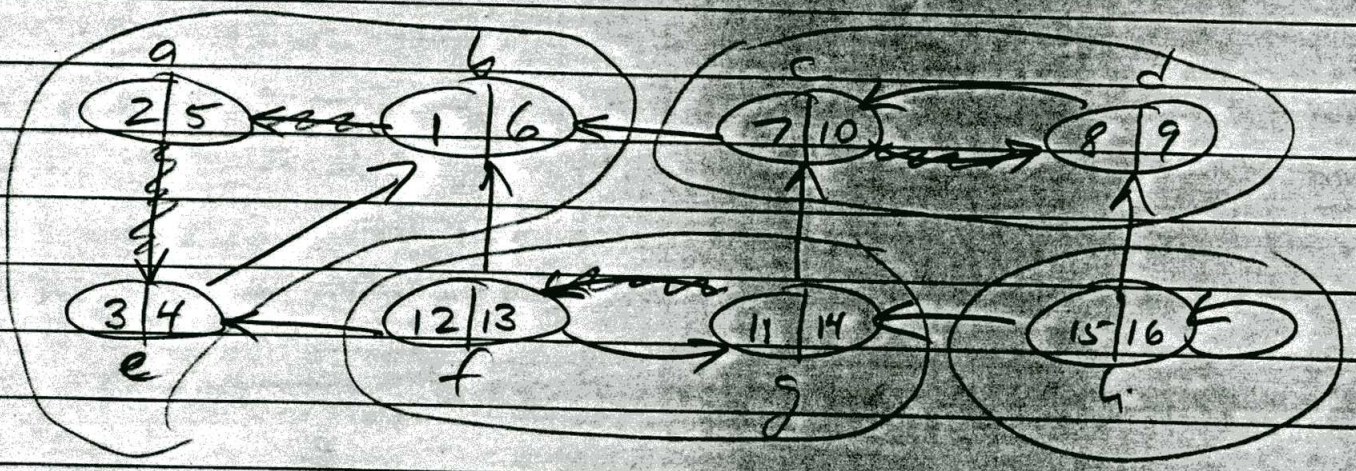
$E = \{(u,v) : (v,u) \in E\}$

Note: G and G^T have same SCC's.

SCC(V,E)

- call DFS(V,E) to compute $f[u]$ for each vertex
- compute G^T
- call DFS(V,E^T), but in main loop of DFS consider vertices in order of decreasing $f[u]$
- output vertices of each ~~DF~~ tree as a separate SCC in the DF forest

G^T:



Correctness: Proof is somewhat involved; see Sec. 23.5. Shorter, incorrect proofs known.

Time: $\Theta(V+E)$. Both DFS calls & G^T computation take $\Theta(V+E)$.

Lectures 6 - Thursday

Goals this week and next ...

Decomposition of Graphs
Depth first and breadth first search

Graphs can be undirected or directed.

Adjacency matrix

$$a_{ij} = \begin{cases} 1 & \text{if there is and edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

For undirected graphs, the matrix is symmetric.
It is $n \times n$ for a graph with $n = |V|$ vertices $v_1, v_2, \ldots, v_n$.
An edge $\{u, v\}$ can be taken in either direction.
Can check if an edge is present in constant time with one memory access.
Takes $O(n^2)$ space which can be wasteful.

Adjacency list

Consists of $|V|$ linked lists, one per vertex.
The list for vertex u contain all v that u has an edge toward. $((u, v) \in E)$
Each edge is in one or two lists.
Total size is $O(|E|)$. (What about O(—V—)?)
Checking for presence of an edge is NOT constant.
Easy to find all the neighbors of a vertex.

Which is best?

For a non-degenerate graph, $|V| \leq |E| \leq |V|^2$.
At the low end, G is *sparse*; at the high end G is *dense*.
The WWW is sparse so a list representation is plausible.

Depth first search

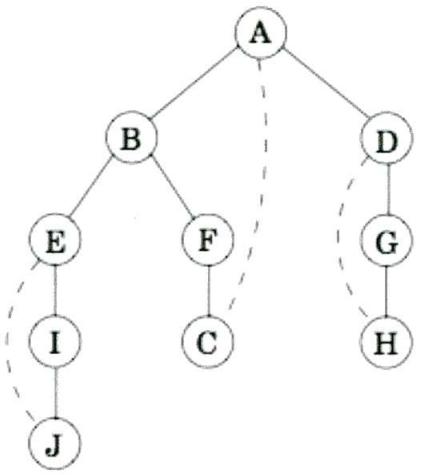
Search a maze with only chalk and a ball of string.

Chalk = Boolean variable "visited"
String unwind to next location = push
String wind back to last location = pop
Recursive version

EXPLORE(G, v)
Input: $G = (V, E)$ is a graph: $v \in V$
Output: visited(u) is set to TRUE for all nodes u reachable from v
visited(v) =TRUE
PREVISIT(v)
for each edge $(v, u) \in E$:
    if NOT visited(u):
        EXPLORE(G, u)
POSTVISIT(v)

PREVISIT and POSTVISIT are optional.
We do get to all vertices that are connected to v.

DFS(G)
for all $v \in V$
    visited(v) = FALSE
for all $v \in V$
    if NOT visited(v)
        EXPLORE(G, v)



Solid edges traversed on a call to EXPLORE that led to discovery of s new vertex.

Solid edges form a tree and are *tree edges.*

Hashed edges led to old stuff. They are *back edges.*

Analysis

1. Each vertex is EXPLOREed just once - thanks to chalk.
   Some amount of work marking spot and PREVISIT and POSTVISIT.
   Total time $O(|V|)$.

2. Loop in which adjacent edges are scanned to see if they must be followed.
   Each edge examined twice $O(|E|)$.

Total time $O(|V| + |E|)$ time it take just to read the list.

Connectivity in undirected graphs

To count *connected components*

PREVISIT(v)
CCNUM[v] = cc

where cc needs to be initialized to zero and to be incremented each time the DFS procedure calls explore. Runs in linear time.

Previsit and postvisit orderings

These two routines will become important.
clock to 1.

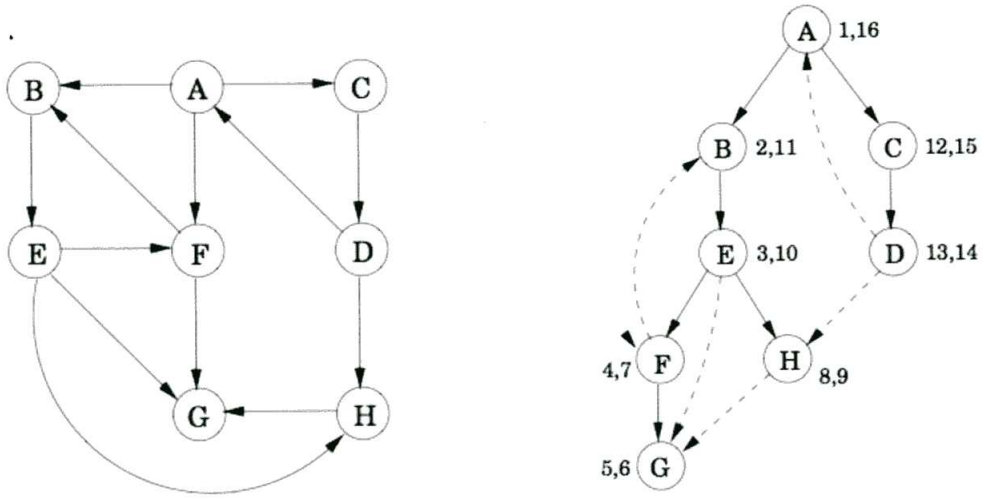
PREVISIT(v)
pre[v] = clock
clock = clock + 1

POSTVISIT(v)
post[v] = clock
clock = clock + 1

Property *For any nodes u and v, the two intervals $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either disjoint or one is contained in the other.*

Proof: $[pre(v), post(v)]$ is the time v is on the stack and stacks are first-in-first-out. □

Figure 3.7 DFS on a directed graph.



A is the *root* of the search tree.
Everything else is its *descendant.*
Ancestors have descendents.
Parents have *children.*

Tree edges are actually part of the DFS forest.
Forward edges lead from a node to a nonchild descendant in the DFS tree.
Back edges lead to an ancestor in the DFS tree.
Cross edges lead to neither descendant nor ancestor; they therefore lead to a node that has already been completely explored (that is, already postvisited).

Ancestor - descendant relationships and type of edge can be read from the PRE and POST values.

Depth-first search strategy $\Rightarrow$ u is an ancestor of v if u is discovered first and v is discovered during EXPLORE(G, u).

$$\begin{matrix} [& [& &] &] \\ u & v & & v & u \end{matrix}$$

PRE/POST **ordering for (u, v) compared to Edge type**

[	[	]	]	Tree/Forward
u	v	v	u	
[	[	]	]	Back
v	u	u	v	
[	]	[	]	Cross
v	v	u	u	

Directed acyclic graphs - dags

A *cycle* in a directed graph is a circular path $v_0 \to v_1 \to \cdots v_k \to v_0 \to$.

A dag without cycles is *acyclic.*

Testing for acyclicity is linear ith a single depth-first search.

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

Proof: $\Leftarrow$ If (u, v) is a back edge, there is a cycle made by this edge plus the path from u to v.

$\Rightarrow$ If the graph has a cycle $v_0 \to v_1 \to \cdots v_k \to v_0 \to$, look at the first node on the cycle to be discovered - the one with the lowest PRE number. All other nodes on the cycle are reachable from it, i.e. are its ~~ancestors~~ descendents but $v_{i-1} \to v_i$ (or $v_k \to v_0$) goes from a node to its ancestor and is then a back edge. □

Directed acyclic graphs - dags can model things like: causalities, hierarhcies, dependencies, prerequisites.

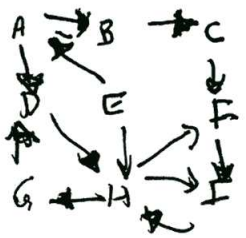
For task dependencies, all tasks leading to a node must be completed before the task can be started. If the graph has a cycle this is impossible. We can always *linearize* or *topologically sort* a dag.

Property *In a dag, every edge leads to a vertex with a lower* POST *number.*

This give a linear time algorithm to linearize a dag: order by decreasing POST number.

source = vertex with no in edges
sink = vertex with no out edges

Property *Every dag has at least one source and at least one sink.*



Linearize fig 3.7 graph.

Strongly connected components.
Connectivity in undirected graphs is easy.

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

See figure 3.9. page 91.

This relation is an equivalence relation. It partitions V into disjoint sets called *strongly connected components.*

Shrink each component to a point and connect these points with an edge in th direction any edges go.

Property *The meta-graph of strongly connected components is always a dag.*

A directed graph is 2-tiered:
A linearizable dag of strongly connected components
A directed graph inside each component.

Efficient algorithm for strongly connected components

Property 1 *If the* EXPLORE *routine is started at node u, then it will terminate precisely when all nodes reachable from u have been reached.*

corollary

If we EXPLORE from a node in a sink component, we will discover exactly the nodes in that component.

Property 2 *The node that receives the highest* POST *number in a DFS must lie in a source strongly connected component.*

This follows from

Property 3 *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'.*

Proof: If DFS visits C before C', then all of C and C' will be traversed before it gets stuck. The first node visited in C wil have the highest POST number.

If DFS visits C' before C, then DFS will get stuck before visiting any of C. So C vertices will have the higher POST numbers. □

But I want a sink ⟸ source in G^R!

the algorithm

1. Run DFS on G^R.
2. Run DFS on G from vertex in 1 with highest POST number. Remove those vertices from G (or their component from the dag) and repeat starting with the remaining vertex highest POST number from step 1.

Show what happens on a graph.