Admin: Cubes out in recitation tomorrow!

Reading: CLRS 22.3–22.4

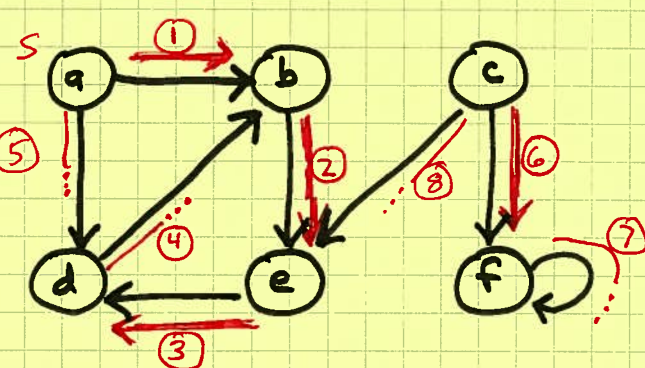
Outline:

- ☐ DFS-review
- ☐ Acyclic directed graphs (DAG's)
- ☐ Topological sort
- ☐ Parallel Processing/critical paths
- ☐ Other searches: time-memory tradeoffs
  simulated annealing

DFS-review:

```
R = set()
for u in V:
    if u not in R:
        R.add(u)
        visit(u)

def visit(u):
    for v in Adj[u]:
        if v not in R:
            R.add(v)
            visit(v)
```



DFS classifies edges: Tree edges (u to child) 1, 2, 3, 6

Non-tree edges:

- Back (u to ancestor) 4
- Forward (u to descendant) 5
- Cross (other: to another subtree) 8
- self-loop 7

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

G is <u>acyclic</u> if it contains no cycles. (≈ DAG, for "directed acyclic graph).

Thm: G is acyclic $\Longleftrightarrow$ DFS of G produces no back edges (or self-loops)

Pf: $\Rightarrow$: if there is a back edge: $u \rightarrow$ ancestor of u, a cycle is created
if self-loop: cycle exists

$\Leftarrow$: Say a vertex "finishes" when its visit call terminates

<u>Lemma:</u> <u>in DFS, all edges (u,v) that are not back edges (or self loop)
have property</u> : <u>v finishes before u</u>

$u \to v$

- tree edge (visit(u) calls visit(v))
- forward edge (visit(v) already done)
- cross edge (")

∴ there are no cycles, since following a path must yield earlier and earlier finishing times...

Proposition: We can tell in $O(V+E)$ time whether a directed graph $G=(V,E)$ is acyclic.

Proof: Run DFS, see if any back edges are produced.

To follow book: use colors:

- all vertices initially white

```
def visit(u):
    color[u] ← gray          //start visiting u
    for v in Adj[u]:
        if v not in R:
            R.add(v)
            visit(v)
    color[u] ← black         //finished!
```

if color[v] = gray: back edge found! G is cyclic!

gray vertices are "in progress", represent current "breadcrumb" path ∎

Application: "Topological Sort"

Given: an acyclic graph $G=(V,E)$

Produce: A list of its vertices

$$v_1 \quad v_2 \quad \text{------} \quad v_n$$

such that all edges go left-to-right.

6.006
Rivest
L14.4
10/20/08

(• Not same as sorting numbers....)

<u>For example:</u>

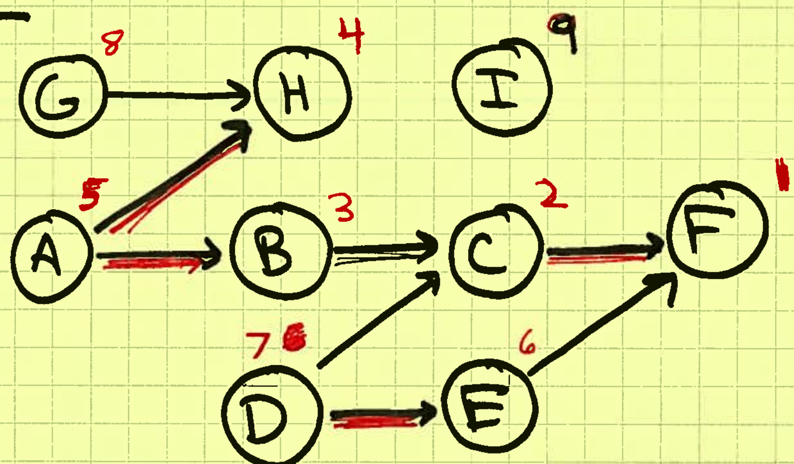
V = set of things to do (tasks)

edge (u, v) means "have to do before you can do v"

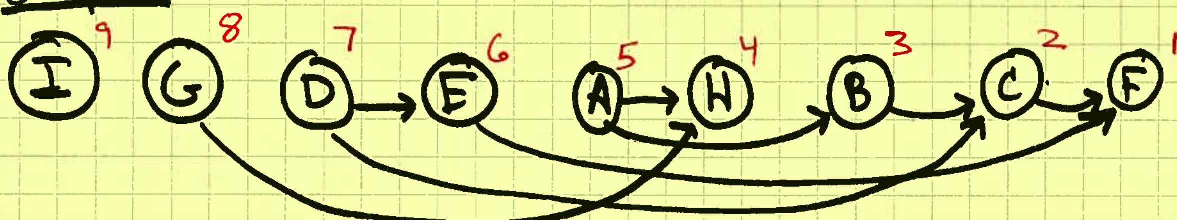
topological sort ⇒ feasible order for doing tasks

<u>Example:</u>

is finishing order



<u>output:</u>



(many other feasible orders exist)

Parallel processing

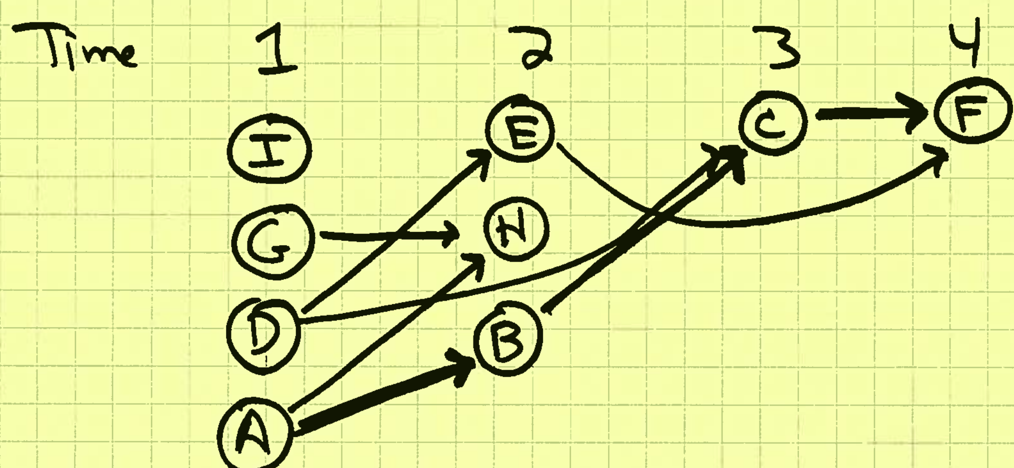
Suppose now we can do many tasks at once (as long as pre-req tasks already done).

How can we find out how long it takes to do all tasks?

(Previous example: 4 time steps
critical path is A → B → C → F)

Idea:

- Produce topological sort
- take elements 1 at a time, put each in earliest time slot possible



(many other solutions exist.)

(Exercise: modify DFS to produce this more directly...)

Other kinds of graph searches:

① time-memory tradeoffs for one-way fn inversion

$$x \longrightarrow f(x)$$

given $f(x)$, how to find x?

- Preprocess graph, store a useful table of info (indep of x, $f(x)$)
- then, given $f(x)$, find x quickly
- But - not enough memory to store all $(x, f(x))$ pairs... ? (cleverness req'd)

② neighborhood search for optimum (simulated annealing)

graph G of states, undirected

"energy" $E(v)$ of vertex v

want to find very low-energy vertex (optimum or near-optimum)

do walk on graph:

pick neighbor of current vertex

if downhill (less energy): go there

else: go up with probability depending on ΔE
and on T (current temperature)

decrease temperature every so often

Pseudocode for simulated annealing:

$T = T_0$ initial temperature

$s = s_0$ start state (initial vertex)

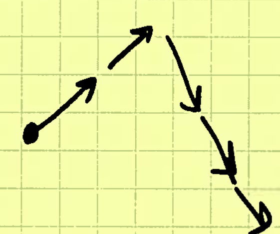
while True:

- $s' =$ random neighbor of s
- $\Delta E = E(s') - E(s)$
- if $\Delta E < 0$: $s = s'$
- else with probability $\exp(-\Delta E / T)$: $s = s'$

every so often: $T = T * 0.99$ (cool down)

As $T \to 0$, uphill moves become less & less frequent

T helps particle "get over hills"



Many variations on these ideas...