lecture 6

Directed Graphs

- Strongly connected components
- Directed Acyclic Graphs (DAG) and Topological Orderings

Background from COMP 250

- graph definitions
  $G = (V, E)$, E represented by
  - adjacency list
  - adjacency matrix

  } or ← TODAY

- graph traversal / search
  - depth first (stack, recursion) ← TODAY
  - breadth first (queue) ← NEXT LECTURE

See my 250 lecture notes (33, 34 + Exercises)

Resources for this lecture

- Roughgarden Algorithms 1 week 4

  https://class.coursera.org/algo-004/lecture/52

  https://class.coursera.org/algo-004/lecture/53

  https://class.coursera.org/algo-004/lecture/54

  } advanced topic (SCC fast)

Sedgewick Algorithm 2 week 1

https://class.coursera.org/algs4partII-002/lecture/11

https://class.coursera.org/algs4partII-002/lecture/10

} lots of nice examples

Undirected graph $G = (V, E)$

- edges are unordered pairs of vertices $e = \{u, v\}$

u • — • v

Directed graph $G = (V, E)$

- edges are ordered pairs of vertices, $e = (u, v)$

i.e. (u, v) is a different edge than (v, u)

} today

u • ⇄ • v

Two vertices $u, v \in V$ in a directed graph are "mutually reachable" if there is a path from u to v and a path from v to u.

u • ⇄ • v

Notes:

- if u and v are mutually reachable, then there is a cycle that contains them
- every vertex is "reachable" from itself by a path of length 0.

Recall from last lecture:

For an undirected graph, we can partition V into "connected components": sets of vertices that are connected by a path.

The corresponding definition for directed graphs is "strongly connected components".

Claim: "mutually reachable" defines an equivalence relation (recall lecture 5)

Proof: Let $\leftrightsquigarrow$ mean mutually reachable. Then its easy to see that:

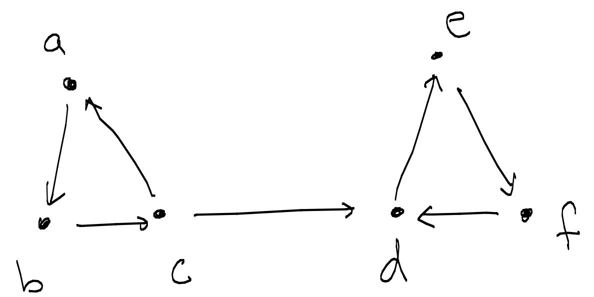
1) $v \leftrightsquigarrow v$ (reflexive)

2) $v \leftrightsquigarrow w \Rightarrow w \leftrightsquigarrow v$ (symmetric)

3) $u \leftrightsquigarrow v$ and $v \leftrightsquigarrow w \Rightarrow u \leftrightsquigarrow w$ (transitive)

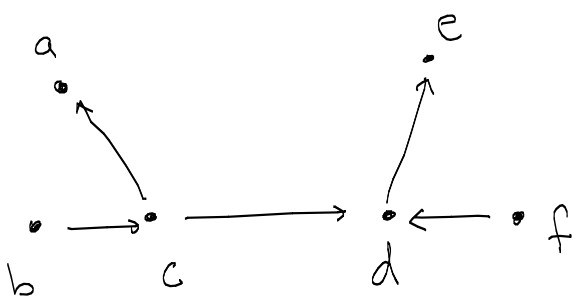
The equivalence classes (i.e. the sets in the partition) are called the "strongly connected components" (SCCs)

Q: what are the SCC's ?



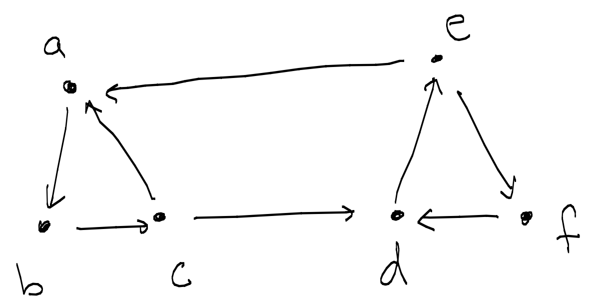
A: $\{a, b, c\}$ $\{d, e, f\}$

Q: what are the SCC's ?



A: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}$

Q: what are the SCC's ?



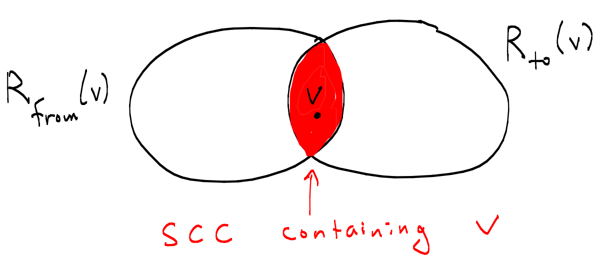
A: $\{a, b, c, d, e, f\}$

Q: Given a graph and a vertex v, how can we find the SCC containing v ?

A: Let R mean 'reachable'.

Let $R_{from}(v) = \{u : v \rightsquigarrow u\}$

$R_{to}(v) = \{u : u \rightsquigarrow v\}$



Q: How to compute $R_{from}(v)$?

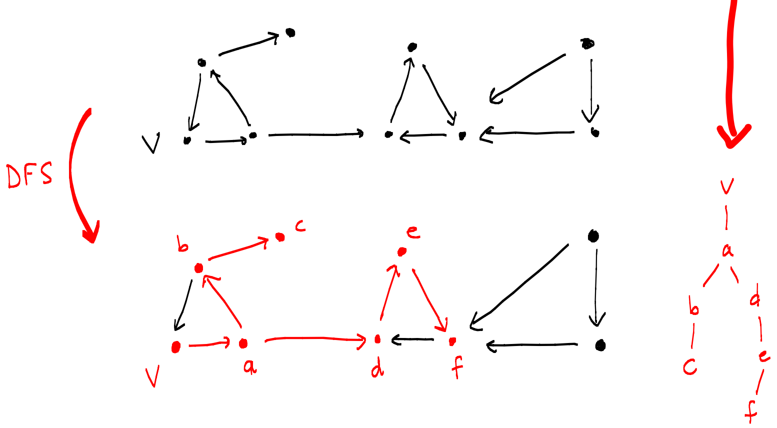
A:

```
DFS(G, v) {
    v.visited = true
    for each w ∈ v.adjList
        if !(w.visited)
            DFS(G, w)
}
```

[BFS also works, of course.]

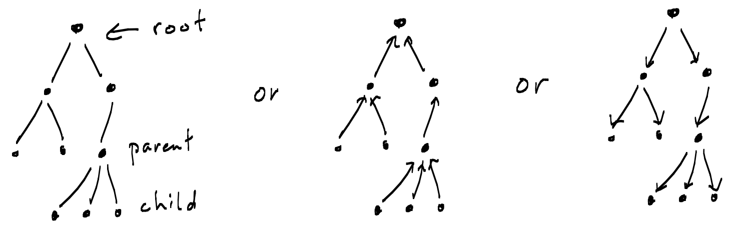
ASIDE: Recall from COMP 250:

DFS(G, v) gives a rooted tree.



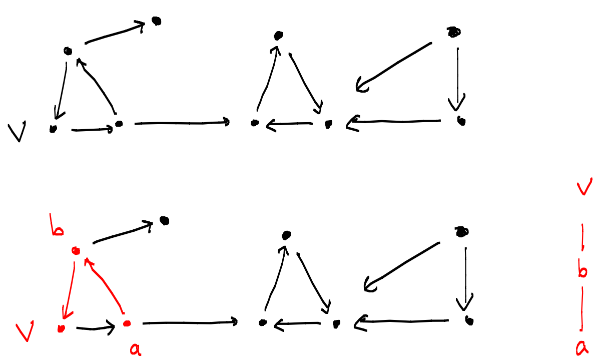
ASIDE:

a "rooted tree" is a tree with one of the vertices specified as the "root". This allows us to talk about parent-child (ancestor - descendent) relationships.



Q: How to compute $R_{to}(v)$?

A: Run DFS(G, v) but follow edges "backwards".



Run DFS(G, v) "backwards" means run DFS(G^T, v) on the reversed graph

$$G^T \equiv (V, E^T)$$

"G transpose"

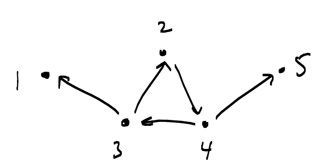
G^T has same vertices as G but the edges are all reversed.

i.e. $(u,v) \in E \iff (v,u) \in E^T$

[Notation: G^T is sometimes called G_{rev} .]

Example :

G

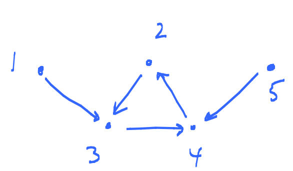


Adjacency Matrix

E

	1	2	3	4	5
1	0	0	0	0	0
2	0	0	0	1	0
3	1	1	0	0	0
4	0	0	1	0	0
5	0	0	0	0	0

G^T



E^T

	1	2	3	4	5
1	0	0	1	0	0
2	0	0	1	0	0
3	0	0	0	1	0
4	0	1	0	0	0
5	0	0	0	0	0

"transpose of E"

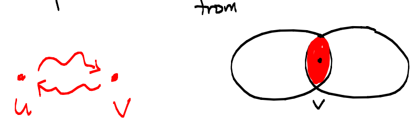
Say it again ...

Q: Given a graph and a vertex v, how can you find the SCC containing v ?

A: DFS (G, v) gives $R_{from}(v)$.

DFS (G^T, v) gives $R_{to}(v)$.

Compute $R_{from}(v) \wedge R_{to}(v)$



Q: Given a graph, how can you find all of the SCC's ?

A:

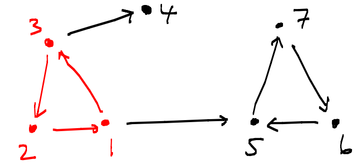
repeat {
- randomly (?) pick a vertex v for which we don't yet know its SCC
- find SCC containing v

}

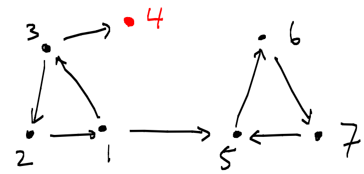
until we know SCC of all vertices

Example

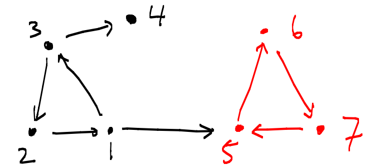
pick 2



pick 4



pick 5



Sadly, this algorithm can be slow.

Example : G is a singly linked list

G $\quad 1 \to 2 \to 3 \to 4 \to 5 \;\cdots\; \to n$

It has n strongly connected components.

$1 \to 2 \to 3 \to 4 \to 5 \;\cdots\; \to n$

$2 \to 3 \to 4 \to 5 \;\cdots\; \to n$

$3 \to 4 \to 5 \;\cdots\; \to n$

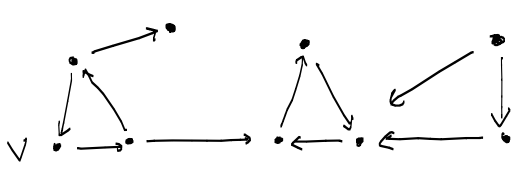
etc.

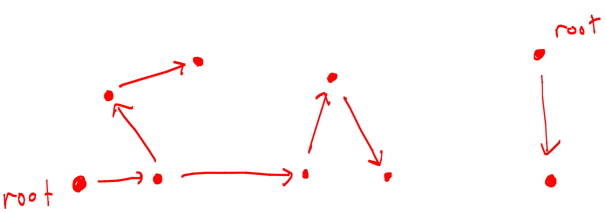
algorithm takes time $\approx n^2$ for this example

There exists a simple algorithm for finding all SCC's in a graph. It is based on:

```
DFS( G ) {
   for all v,
       v.reached = false
   while there exists v such that !(v.reached)
         DFS( G, v )
}
```

DFS(G) gives a forest of DFS trees.





Fast Algorithm for finding all SCCs

1) Run DFS(G^T)

2) Run DFS(G)

or swap order (doesn't matter)

NOT ON EXAM!

The clever trick [Kosaraju 1978] is how to choose v in the while loop of DFS(G).
(not random!)

Sedgewick Alg. 2 → https://class.coursera.org/algs4partII-002/lecture/11

takes Roughgarden ~1 hour to go over details / example / proof

https://class.coursera.org/algo-004/lecture/53

https://class.coursera.org/algo-004/lecture/54 lecture 6

Directed Graphs

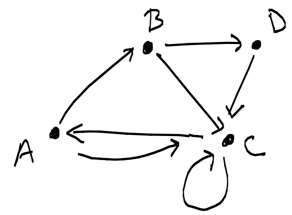
- Strongly connected components
- Directed Acyclic Graphs (DAG) and Topological Orderings

Directed Acyclic Graph (DAG)

- directed graph that has no cycles

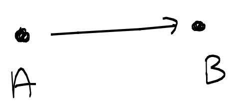
[a cycle is a sequence of vertices such that the first vertex is the same as the last vertex]

Example



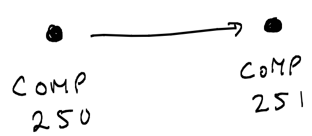
cycles	not cycles
C C	C
A C A	A C
A B C A	B C D B
B D C A B	B D C B
⋮	⋮

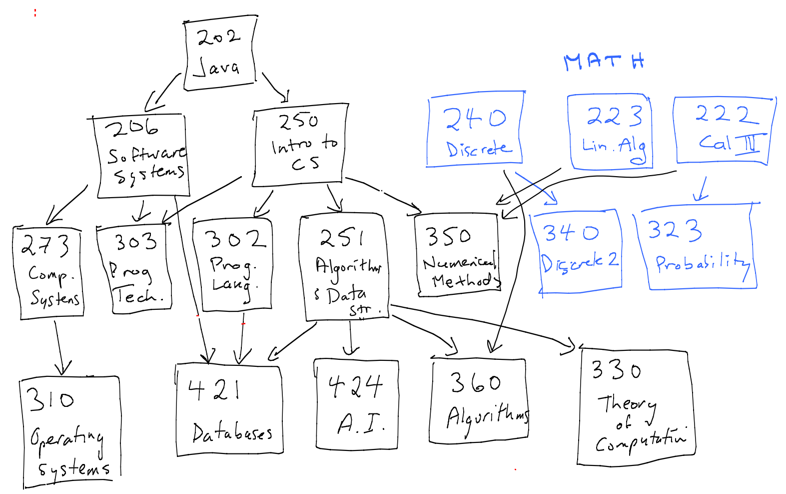
DAGs are often used to capture dependencies between events.



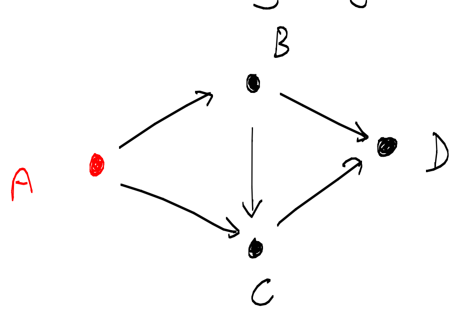
e.g. could represent that B can only occur once A has occured.

e.g. prerequisities relation





Claim: If G is a DAG, then G must have at least one vertex with no incoming edges.

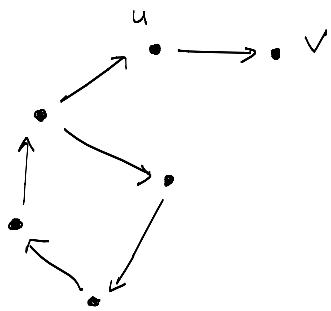


[Exercise: if G is a DAG, then G must have at least one vertex with no outgoing edges.]

Proof: (by contradiction)

Suppose every vertex of our DAG has at least one incoming edge. Pick any vertex v. v has an incoming edge (u,v), so follow it backwards to u. u has an incoming edge, so follow it backwards, etc. Eventually we must reach a vertex we have visited already (since there is a finite # vertices). But this would define a cycle.

Note the cycle might not involve u, v.



lecture 6

Directed Graphs

- Strongly connected components
- Directed Acyclic Graphs (DAG) and Topological Orderings

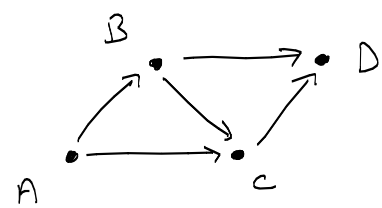
"Topological Ordering"

Given a graph $G = (V, E)$, label the vertices $v_1\ v_2 \ldots v_n$ such that:

if $(v_i, v_j) \in E$ then $i < j$.

This vertex ordering (when it exists!) is called a "topological ordering".

<u>Example</u>



$v_1\ v_2\ v_3\ v_4$
A B C D is a topological ordering.

A C B D is <u>not</u>. Why not?

<u>Claim:</u> If $G = (V, E)$ is a DAG, then G has a topological ordering.

<u>Proof:</u> (constructive)

$G' = G$ // n vertices, $V' = V$, $E' = E$

for $i = 1$ to n {

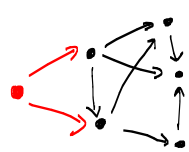
v_i = a vertex in G' with no incoming edges

$V' = V' \setminus \{v_i\}$

$E' = E' \setminus \{(v_i, w) \in E'\}$

}

"set difference"



<u>Claim:</u> If a directed graph G has a topological ordering, then G is a DAG.

i.e. G doesn't have a cycle.

<u>Equivalent Claim:</u> If a directed graph G has a cycle, then G does not have a topological ordering.

ASIDE: Logic : Contrapositive statement (MATH 240)

statement		contrapositive
"p implies q"		not(q) implies not(p)
"if p then q"	logically equivalent	"if not(q) then not(p)"
$p \Rightarrow q$		$\text{not}(q) \Rightarrow \text{not}(p)$

http://en.wikipedia.org/wiki/Contraposition

Equivalent Claim: If a directed graph G has a cycle, then G does not have a topological ordering.

Proof: (by contradiction)

Let the cycle be $(v, \ldots, v)$.

If there were a topological ordering then we could write the cycle as

$$(v_i \; \overbrace{v_j \quad v_k \quad \cdots}^{\text{possibly empty}} \; v_i)$$

where $(v_i\, v_j)$, $(v_j\, v_k)$, etc. are edges in E and $i < j < k < \ldots < i$.

But then $i < i$ which is obviously false.

Announcements

- A1 due on Sunday night
- my office hours
  Tues & Thurs.
  11:30 – 12:30
  + 1:00 - 2:00 pm starting next week
- Code for arraylist, linked lists, BSTs, hash table available from my COMP 250 page