CS II Lecture 5

- Finding Strongly Connected Components in a directed graph
- Breadth First Search

Finding SCCs

We have

- An algorithm that finds connected components in undirected graphs
    - what does it do in directed graphs?
- An algorithm that finds a topological sort in directed acyclic graphs
    - what does it do in directed graphs with cycles?

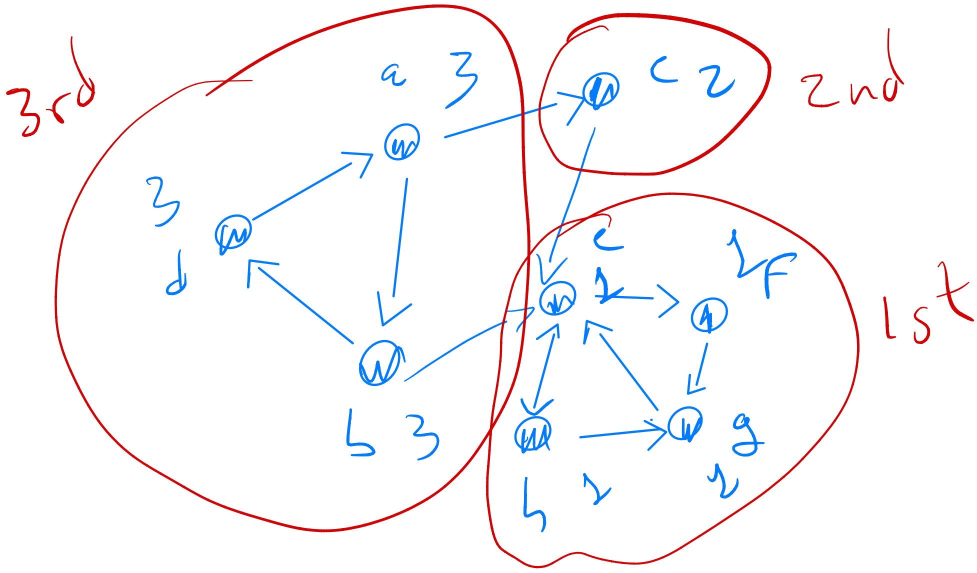
- An algorithm that finds connected components in undirected graphs

```
Global variables:
    graph G = (V, E)
    array visited[ ] one entry per vertex
        initialized to F

    array comp[] of integers, one
        entry per vertex

def explore(v, c)
    visited[v] = T ; comp[v] = c
    for each w such that (v,w) ∈ E :
        if not visited[w]:
            explore(w, c)

def DFS_cc( )
    c = 0
    for each v in V:
        if not visited[v]:
            c = c + 1
            explore(v, c)
```



Run using

a, b, c, d, e, f, g, h

as order in "for each v in V"

Run using

e, f, b, h, a, c, d, g

Run using e, f, c, h, b, a, d, g

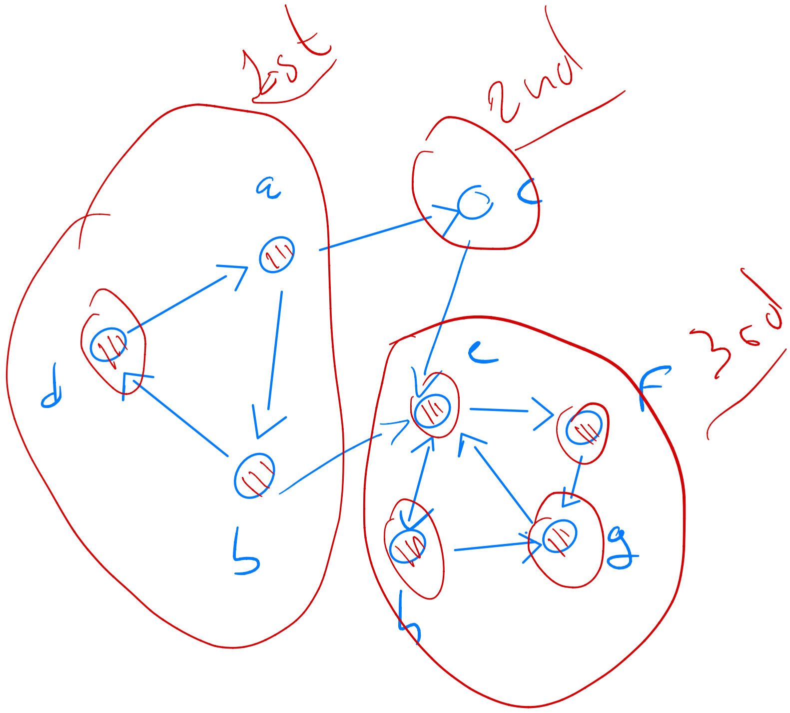
Ok if order in which SCCs first appear in list is reverse of topological order among SCCs

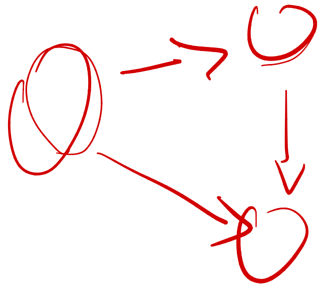
An algorithm that finds a topological sort in directed acyclic graphs

```
Global variables:
        graph G = (V, E)
        array visited[ ]   one entry per vertex
              initialized to F
        list L initialized to ∅

def explore(v):
        visited[v] = T
        for each w such that (v, w) ∈ E:
                if not visited[w]:
                        explore(v)
        add v at head of L


def DFS():
        for each v in V:
                if not visited[v]:
                        explore(v)
```





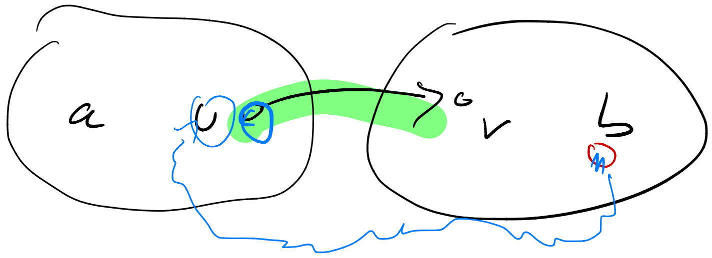
Using a, b, c, d, e, f, g, h order
in "for each v in V"

L = a c b e h f g d

SCCs first appear in list
in topological order of SCCs

Lemma -- a...b..u..v ---

-- a..u..b...v



If (u,v) is an edge between SCCs, a is last one to complete explore(·) in SCC of u, b is last one to complete explore(·) in SCC of v, then explore(a) terminates after explore(b)

Proof

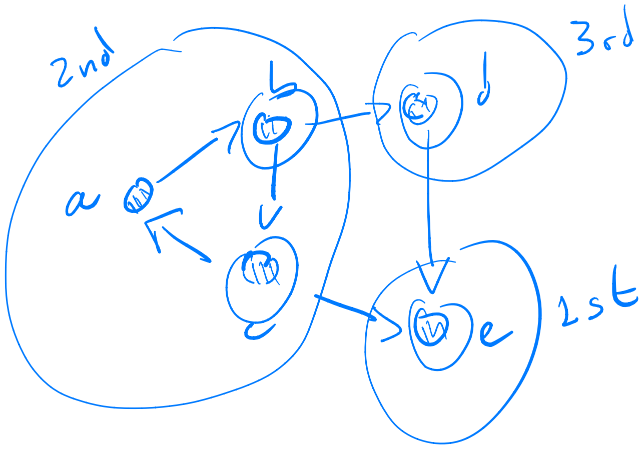
when explore(u) looks at (u,v)

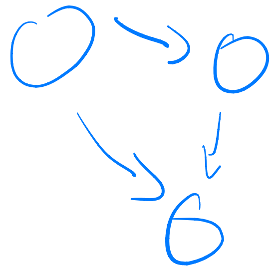
visited[b] = F	visited[b] = T
visited[v] = F discovers b inside explore(u) explore(a) terminates after explore(u) explore(u) terminates after explore(b)	explore(b) has terminated while explore(u) active

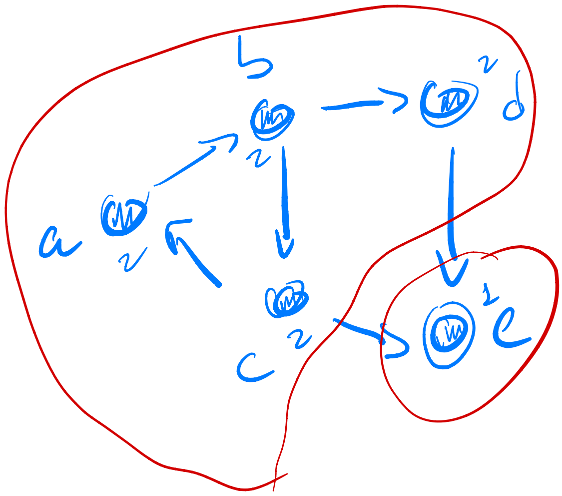
Algorithm for SCCs

Given G

- Reverse direction of edges to create G^R
- Run topological sort algorithm on G^R
  - obtain list L of vertices such that order in which SCCs of G^R first appear in G^R is topological order of SCC
  - SCCs of G appear in L in reverse of topological order of G
- Run undirected CC algorithm on G using L for order in which to run "for each v in V"

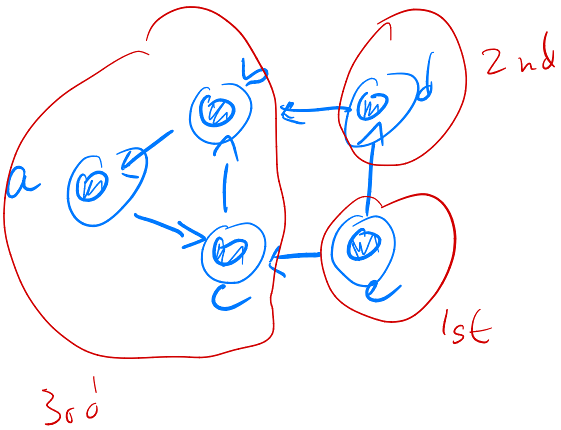




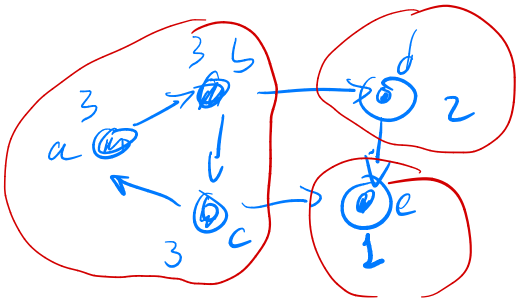


a b d c e

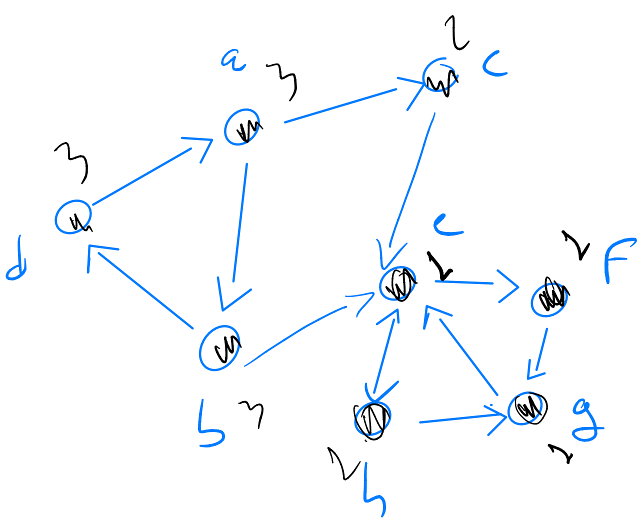
e c d b a



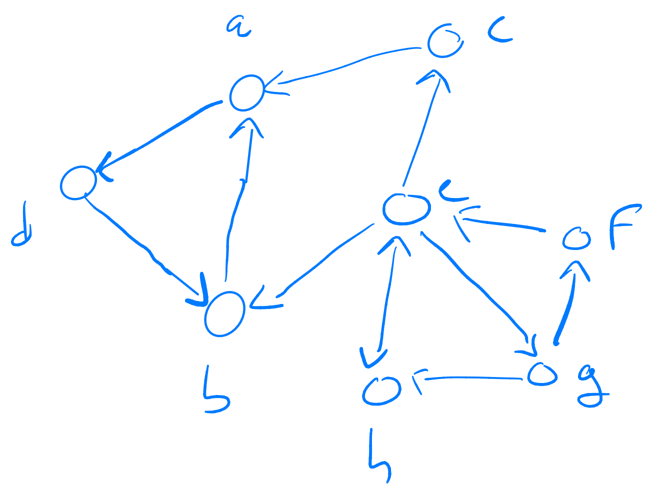
e d a c b



Example



G

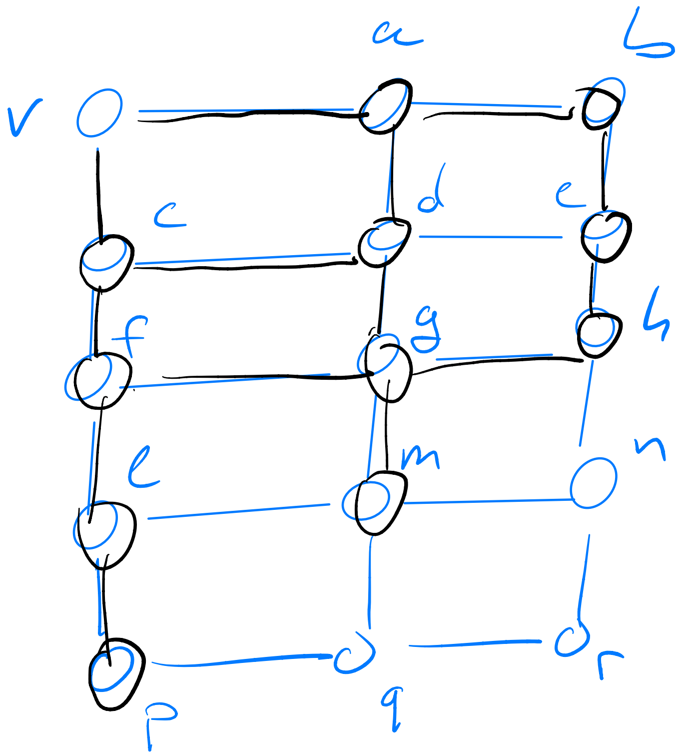


G^R

- Run T.S. algorithm on G^R
  obtain list
  $$L = \underline{e}\ \underline{g}\ \underline{h}\ \underline{f}\ \underline{c}\ \underline{a}\ d\ b$$

- Run undirected cc algorithm on G using L for "for each v in V" loop

New goal: explore a graph starting from a node v in increasing distance from v



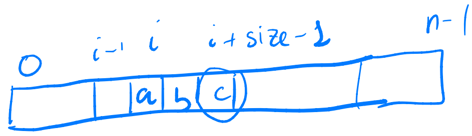
v, a, c, b, d, f, e, g, l

Queue

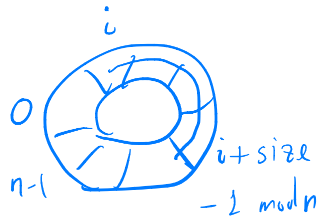
operations

- create empty queue
- insert an element
- remove and read "oldest element"

insert(b)
insert(c)
insert(a)
remove() = b
insert(d)
remove() = c



size = 3
first = i



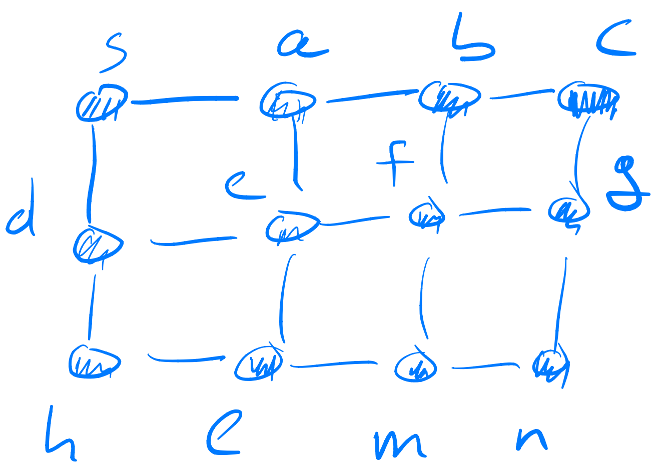
Recall queue:

data structure that supports operations of

- insert an element
- remove and read "oldest element"

```
def BFS(G, s):
    visited[] = boolean array indexed by
                vertices initialized to F

    Q = initially empty queue
    visited[s] = T
    Q.insert(s)
    while not Q.empty():
        v = Q.eject()
        for each (v,w) edge in G:
            if not visited[w]:
                visited[w] = T
                Q.insert(w)
```



Q	visited
s	s
a, d	s, a, d
e, h, a	s, a, d, e, h
b, e, h	s, a, d, e, h, b

Q

order in which nodes come out of Q

s, d a e h, b, f, l, c, g m, n

$\underbrace{s}_{}\ , \underbrace{d\ a}_{1}\ \underbrace{e\ h,\ b}_{2},\ \underbrace{f,\ l,\ c}_{3},\ \underbrace{g\ m}_{4},\ \underbrace{n}_{5}$

Properties of BFS

- nodes are removed from Q in increasing distance from s

- edge (u, v) that causes v to be added to Q is in shortest path from s to v

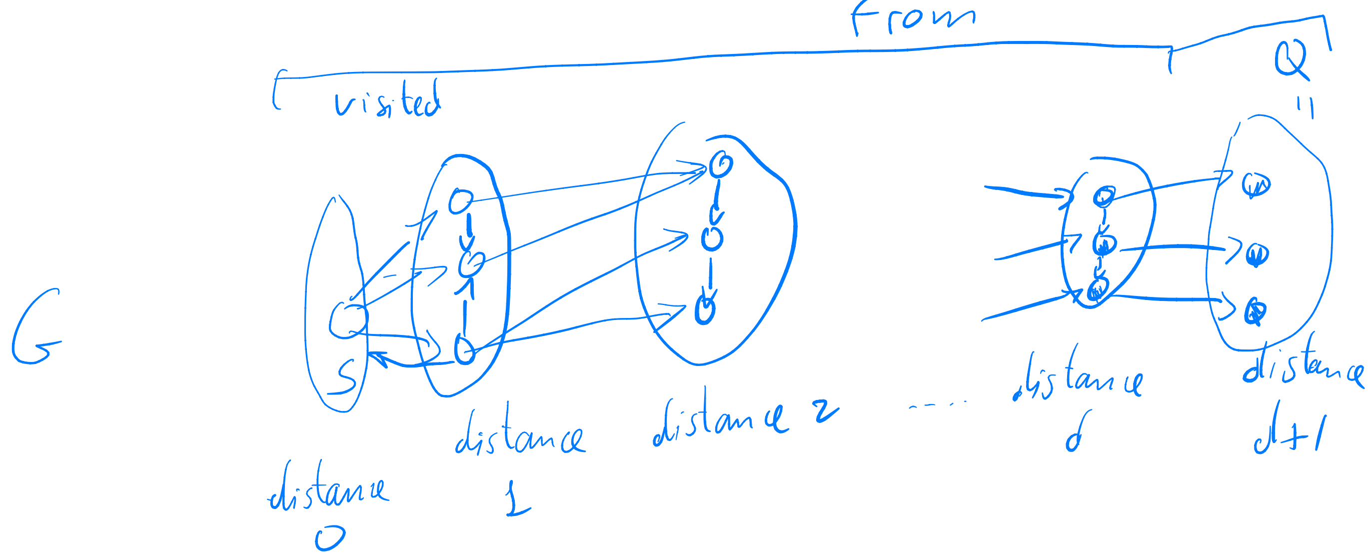
Lemma

For each $d = 0, 1, 2, \ldots$ there is a point in time in the execution of the algorithm such that the visited nodes are precisely those at distance $\leq d$ from s, and Q contains precisely the vertices at distance $= d$

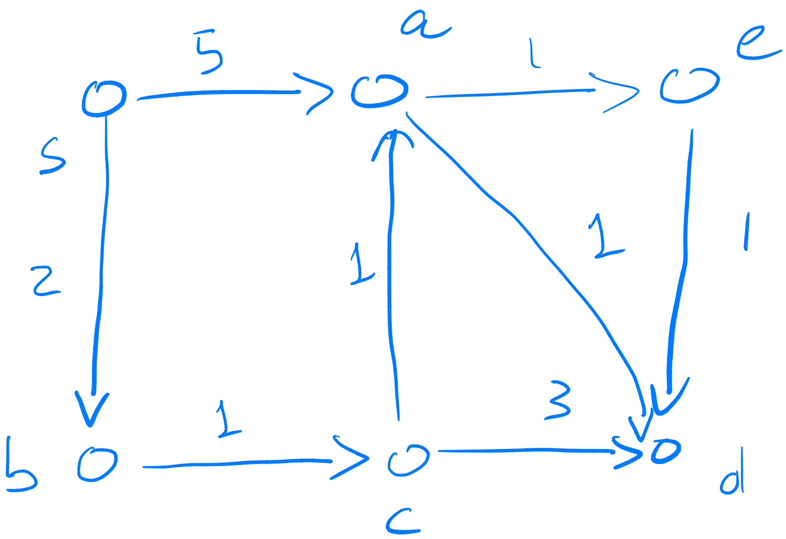
$d = 0$ happens before start while loop

---

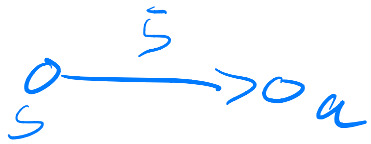
suppose there is point in time when
Q contains precisely vertices at distance d
visited[v] = T iff v has distance $\leq d$ from



Ideas of Dijkstra



Simulate BFS in which



is replaced by