Graphs

It’s all about relationships

Terence Parr
MSDS program
University of San Francisco

UNIVERSITY OF SAN FRANCISCO

What's a graph?

- A graph is a collection of connected element pairs
- As with a tree, a graph is an aggregate of nodes
- Elements/nodes are email addresses, map locations, documents, tasks to perform, URLs on the web, customers, computers on a network, friends, observations, sensors, states in markov chain, …
- Terms: *nodes* or *vertices* connected with *edges*, which can have labels; e.g., recall the Trie graph with labeled edges
- *Directed* graphs have arrows as edges but *undirected* use lines
- For *n* nodes, num edges is >=0 and <= $\binom{n}{2}$ = $n(n-1)/2$

Common questions

- Is q reachable from p?
- How many edges are on paths between p and q?
- Is graph connected? (reach any p from any q)
- Is graph cyclic? (p reaches p traversing at least one edge)
- Which nodes are within k edges of node p? (neighborhood)
- What is shortest path (num edges) from p to q?
- What is shortest path using edge weights? [beyond scope of 689]
- Traveling salesman problem [beyond scope of 689]

Adjacency matrix implementations

- Adjacency matrix, n x n matrix of {0,1} if unlabeled or {labels} if edges are labeled; undirected matrices are symmetric
- Wastes space for sparse edges; use sparse matrix
- Fast to access arbitrary node's edges

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*
a	0	1	0	0	1	0	0	0	0	0	0	0	0
b	1	0	1	0	1	1	0	0	0	0	0	0	0
c	0	1	0	1	0	1	1	0	0	0	0	0	0
d	0	0	1	0	0	0	1	0	0	0	0	0	0
e	1	1	0	0	0	1	1	1	0	0	0	0	0
f	0	1	1	0	1	0	1	0	0	0	0	0	0
g	0	0	1	1	1	1	0	0	1	0	0	0	0
h	0	0	0	0	1	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	1	0	0	0	0	0	0
j	0	0	0	0	0	0	0	0	0	0	1	1	1
k	0	0	0	0	0	0	0	0	0	1	0	1	0
l	0	0	0	0	0	0	0	0	0	1	1	0	1
m	0	0	0	0	0	0	0	0	0	1	0	1	0

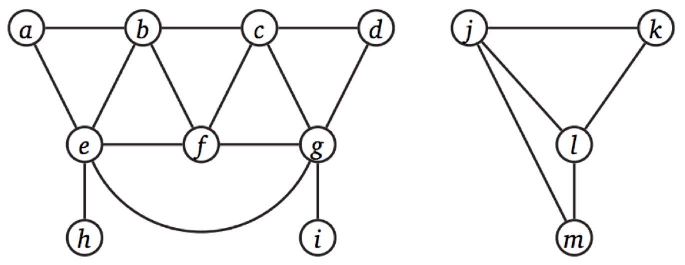


Figure 5.11. An adjacency matrix for our example graph.

http://jeffe.cs.illinois.edu/teaching/algorithms/book/05-graphs.pdf

Adjacency list implementations

- List of edge lists for nodes
- Fast arbitrary node access for numbered nodes, space efficient

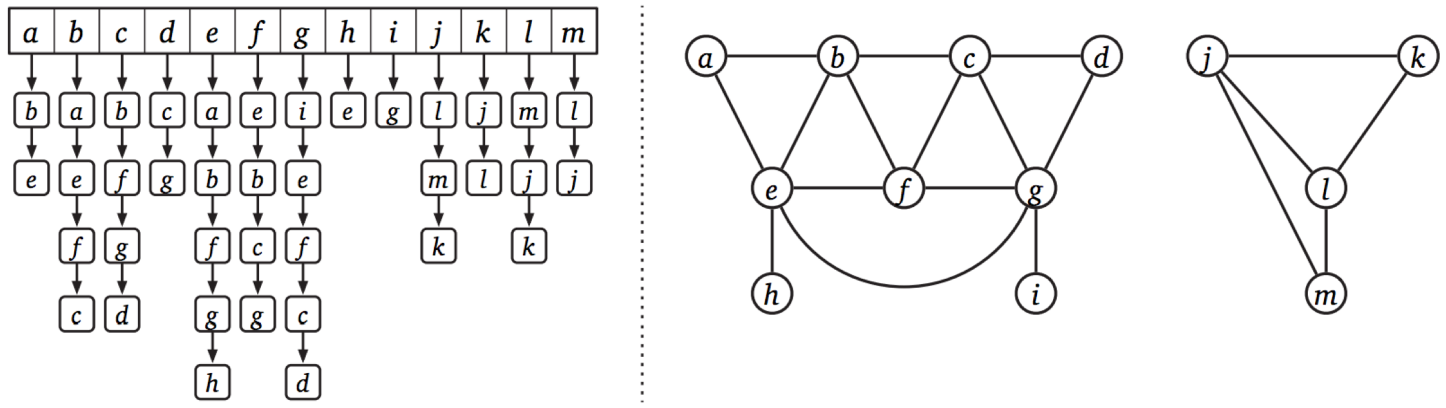


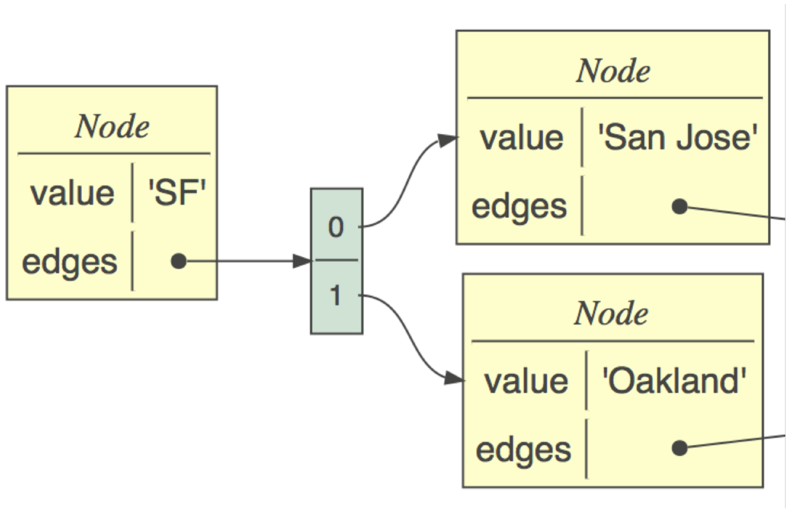
Figure 5.9. An adjacency list for our example graph.

http://jeffe.cs.illinois.edu/teaching/algorithms/book/05-graphs.pdf

Connected nodes implementation

- Most common implementation due to nice mapping to objects
- Each node has info about node and edge list
- Use list or dictionary index if you need to access nodes directly

```
class Node:
  def __init__(self, value):
    self.value = value
    self.edges = []
  def add(self, target:Node):
    self.edges.append(target)
```

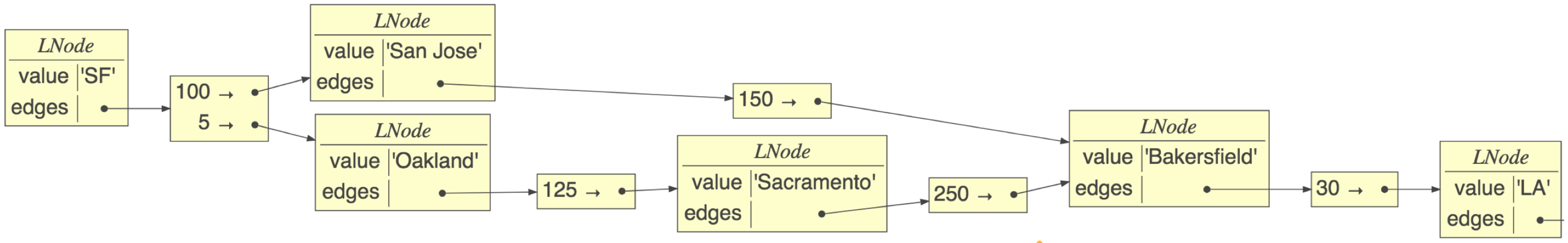


Implementation with labels

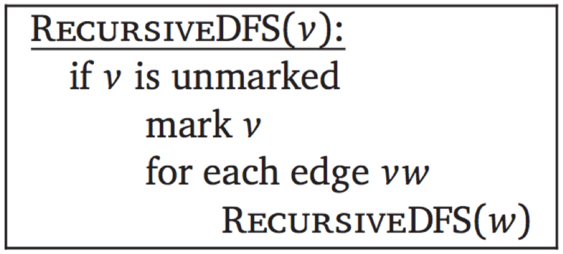
```
class LNode:
  def __init__(self, value):
    self.value = value
    self.edges = {}
  def add(self, label, target):
    self.edges[label] = target
```

Edge->node dictionary, not list

```
sf.add(100,sj)
sj.add(150,baker)
…
```



Depth-first search (review)



```
RecursiveDFS(v):
    if v is unmarked
        mark v
        for each edge vw
            RecursiveDFS(w)
```

- The fundamental algorithm for answering graph questions
- Visits all reachable nodes from p, avoiding cycles

```
def walk_graph(p:Node, visited=set()):
    if p in visited: return
    visited.add(p)
    for q in p.edges:
        walk_graph(q, visited)
```

O(n,m) = n + m, for n nodes, m edges; m can be n^2

Is there a cycle from p to p?

- If we run into starting node in visited set, return True

```
def iscyclic(p:Node) -> bool:
    return iscyclic_(p,p,set())

def iscyclic_(start:Node, p:Node, visited) -> bool:
    if p in visited:
        if p is start: return True # we find start?
        return False # can't loop forever so stop
    visited.add(p)
    for q in p.edges:
        c = iscyclic_(start, q, visited)
        if c: return True # we can stop
    return False
```

Find set of nodes p can reach

- Need two sets, one for avoiding cycles, another for reached nodes

```
def reachable(p:Node) -> set:
    reaches = set();
    reachable_(p, reaches, set())
    return reaches

def reachable_(p:Node, reaches:set, visited:set):
    if p in visited: return
    visited.add(p)
    for q in p.edges:
        reaches.add(q)
        reachable_(q, reaches, visited)
```

Find set of nodes p can reach, track depth

- Track node->depth map, not just set of nodes

```
def reachable(p:Node) -> dict:
    reaches = dict()
    reachable_(p, reaches, set(), depth=0)
    return reaches

def reachable_(p:Node, reaches:dict, visited:set, depth:int):
    if p in visited: return
    visited.add(p)
    reaches[p] = depth
    for q in p.edges:
        reachable_(q, reaches, visited, depth+1)
```

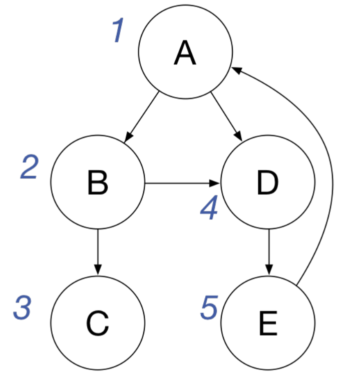
Find neighborhood within k edges

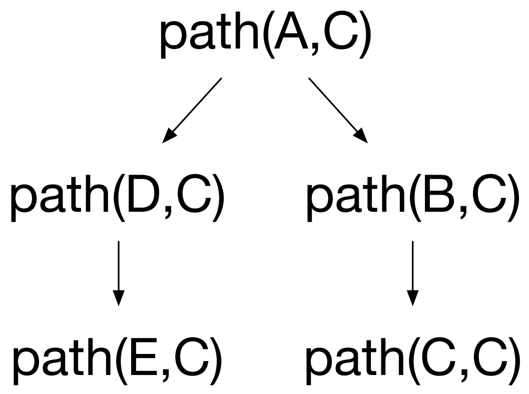
- Track dict node->depth, stop when we reach depth

```
def neighbors(p:Node, k:int) -> dict:
    reaches = dict()
    neighbors_(p, k, reaches, set(), depth=0)
    return reaches

def neighbors_(p:Node, k:int, reaches:dict, visited:set, depth:int):
    if p in visited or depth>k: return
    visited.add(p)
    reaches[p] = depth
    for q in p.edges:
        neighbors_(q, k, reaches, visited, depth+1)
```

Find path from p to q

```
def path(p:Node, q:Node) -> list:
    return path_(p, q, [p], set())

def path_(p:Node, q:Node, path:list, visited:set):
    if p is q: return path
    if p in visited: return None
    visited.add(p)
    for t in p.edges:
        pa = path_(t, q, path+[t], visited)
        if pa is not None: return pa
    return None
```



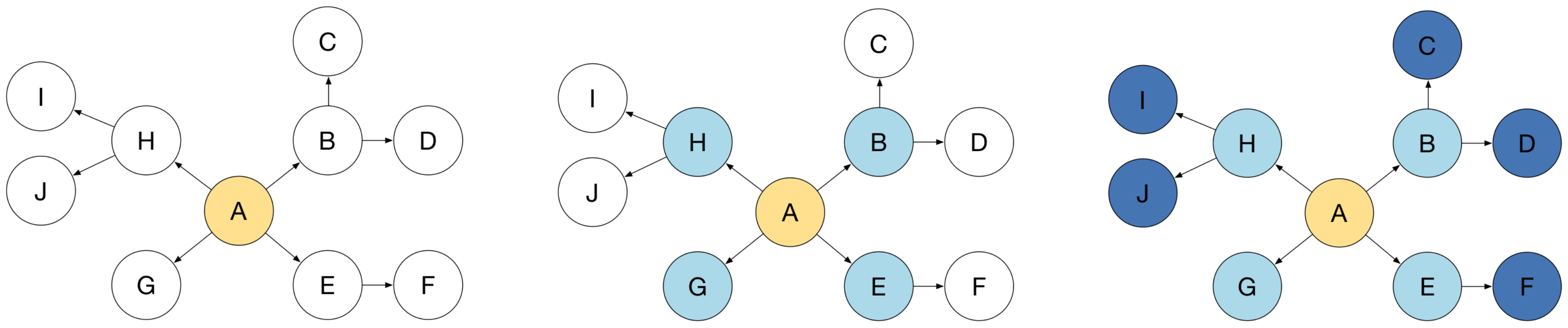


recursion tree

Must track path not set of nodes

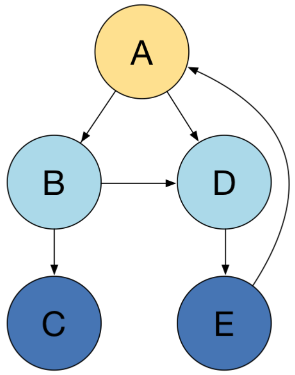
Breadth-first search vs DFS

- Visit all children then grandchildren…



UNIVERSITY OF SAN FRANCISCO

BFS implementation



- Maintains work list of nodes and visited set
- **BFS** **DFS**

BFS	DFS
Visit A	Visit A
Visit B	Visit B
Visit D	Visit C
Visit C	Visit D
Visit E	Visit E

- Add to work list end, pull from front (queue)

```
def BFS(root:LNode):
    visited = {root}
    worklist = [root]
    while len(worklist)>0:
        p = worklist.pop(0)
        print(f“Visit {p}”)
        for q in p.edges:
            if q not in visited:
                worklist.append(q)
                visited.add(q)
```

Find **shortest** path from p to q?

$$\begin{tikzcd} A \arrow[r] \arrow[rr, bend left] & B \arrow[r] \arrow[d] & D \arrow[d] \\ & C & E \end{tikzcd}$$

- BFS where work list is a list of paths not list of nodes
- Tail of path is where we left off work on it
- By searching all children before going deeper, we auto-magically find paths with shortest lengths

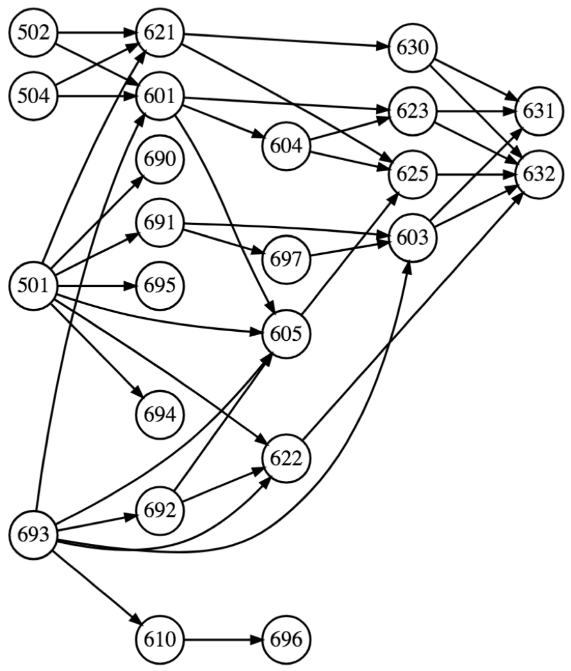
```
def shortest(root:Node, target:Node):
    visited = {root}
    worklist = [[root]]
    while len(worklist)>0:
        path = worklist.pop(0)
        p = path[-1] # tail of path
        if p is target: return path
        for q in p.edges:
            if q not in visited:
                worklist.append(path+[q])
                visited.add(q)
```

https://github.com/parrt/msds689/blob/master/notes/graphs.ipynb

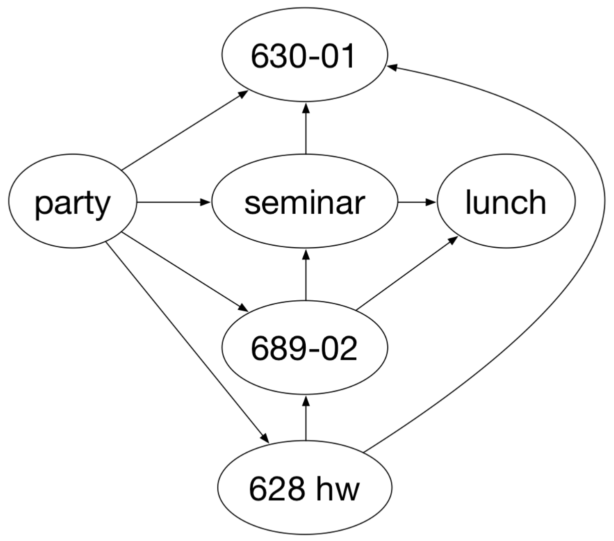
UNIVERSITY OF SAN FRANCISCO

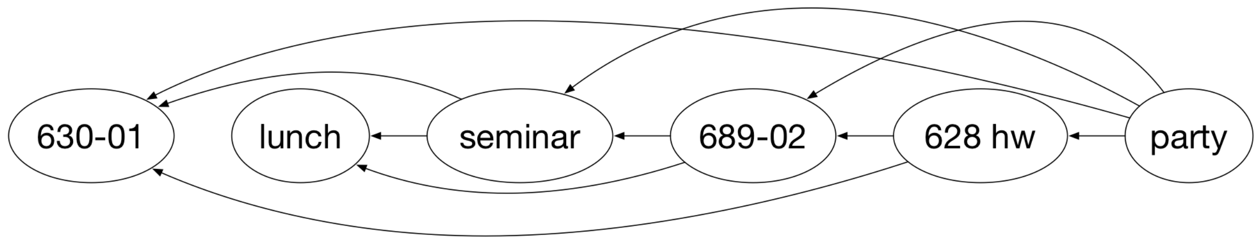
Topological sort (acyclic graphs)

- **Problem**: Find linear ordering of nodes in directed acyclic graph such that all constraints, u->v, are satisfied where u depends on v so v must come before u
- Examples: task ordering or course prereq chain.
- Can also have u->v mean u precedes v. E.g., 502 is prereq for 621 and 601…
  Find order we should take classes
- Sort is not usually unique



Example topological sort

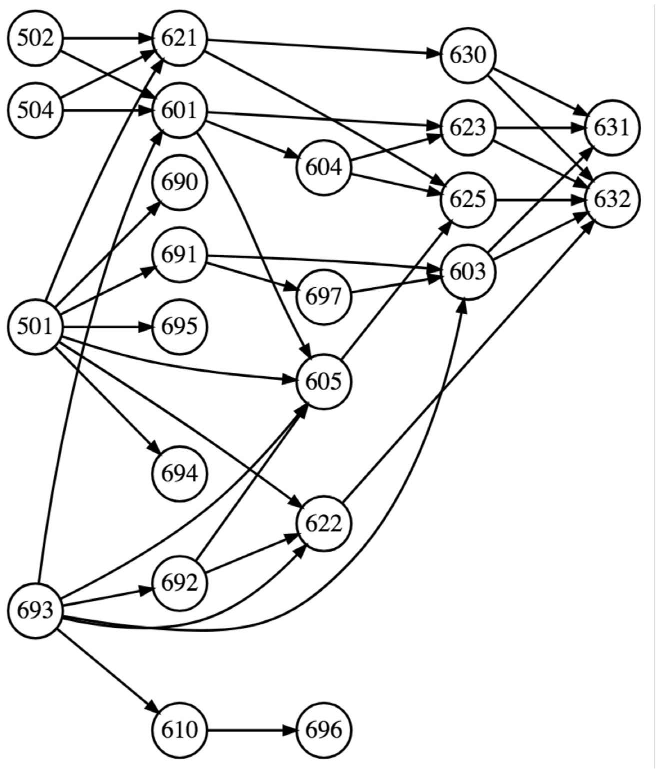


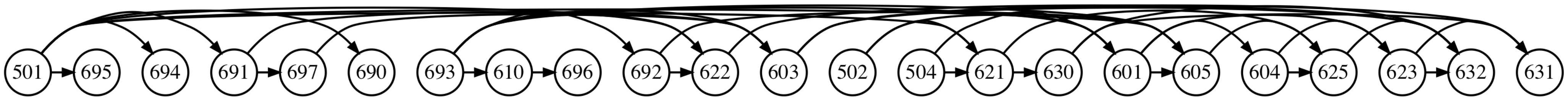


If u depends on v, any linear ordering where edges point to left is solution

Example where u precedes v

- In this case, edges must point to the right

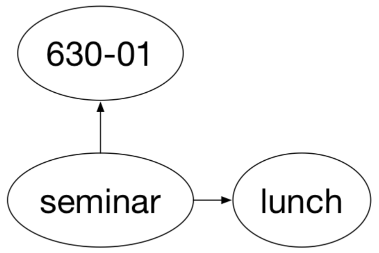




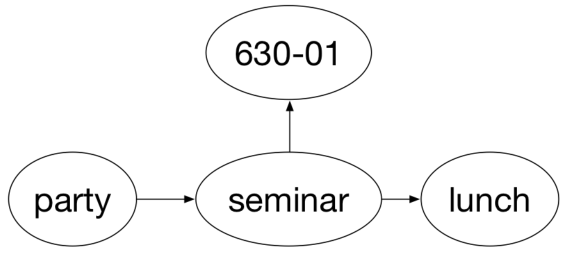
UNIVERSITY OF SAN FRANCISCO

How to approach the problem

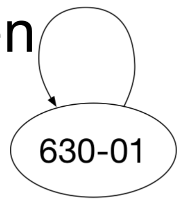
- What order should we do these tasks (u depends v)? Think in terms of traversal order



- What if we add party goal?



- BTW, cycles would cause trouble; are meaningless when dealing with dependencies; how can 630-01 be attended before itself?

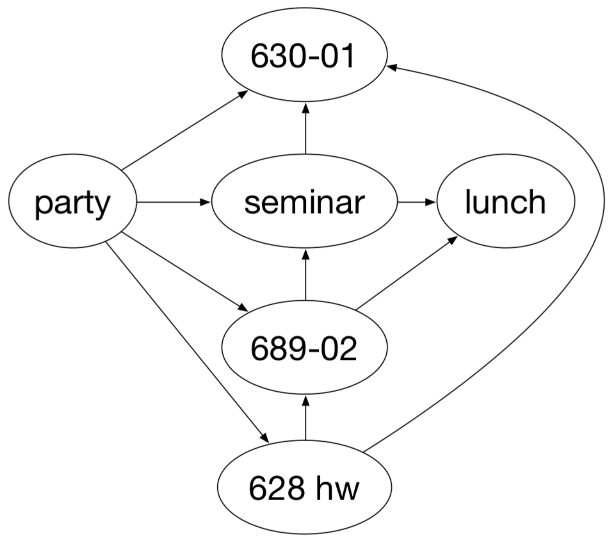


UNIVERSITY OF SAN FRANCISCO

DFS-based topo sort implementation

- Lots of very complex algorithms on the web
- Simplest solution: DFS-based topological sort
- A valid sort is just the post-order graph traversal if u depends v!
- If u precedes v, reverse the post-order traversal
- Well, we have to make sure to do DFS on all root nodes (nodes w/o incoming edges) but core is just DFS
- With one root, it’s just postorder traversal via DFS

Example walk through



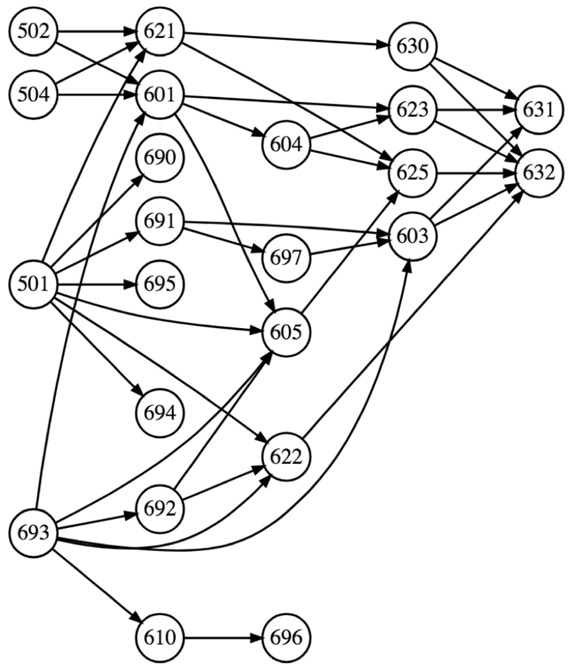
- DFS starting with party:
  party -> 628 -> 630
  back out then hit 689 then lunch
  back out and hit seminar
- Postorder traversal processes **after** visiting children:
  630, lunch, seminar, 689, 628, party
- Solution: 630-01, lunch, seminar, 689-02, 628 hw, party

DFS postorder traversal

```
def postorder(p:Node, sorted:set, visited:set):
    if p in visited: return
    visited.add(p)
    for q in p.edges:
        postorder(q, sorted, visited)
    sorted.append(p)
```

With multiple roots, hit them all

```
def toposort(nodes):
    sorted = []
    visited = set()
    while len(visited) < len(nodes):
        todo = [node for node in nodes.values()
                if node not in visited]
        if len(todo)>0:
            postorder(todo[0], sorted, visited)
    return sorted
```



UNIVERSITY OF SAN FRANCISCO

Summary

- Graphs are for showing relationships between elements
- DFS for finding a path or multiple paths or cycles
- BFS for find shortest (in edges) path or neighborhood
- DFS postorder great for topo sort
- Recursive alg's all use **visited** set to avoid cycles
- Non-recursive DFS: (use work list stack)
    - push targets in reverse order onto work list
    - pop last work list item for next node to process
- Non-recursive BFS: (use work list queue)
    - push targets in order onto work list
    - pull from first position

Sample graph problems

UNIVERSITY OF SAN FRANCISCO

Exercise

- Given a directed graph, detect all direct or indirect cycles
- For p in nodes: report iscyclic(p)

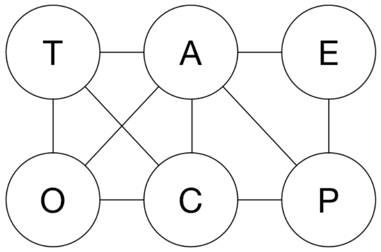
Exercise

- Given 2 lists P,Q and function conn(p,q)=true if edge p->q exists. P is origin (starting) nodes and Q destination nodes. Report 1 for P[i] reaches Q[i] directly or indirectly.
- Create graph using conn(p,q) for all nodes in P and Q
- For each P[i], see if Q[i] is in reaches(P[i]) set.

Exercise: Boggle

- Given m x n matrix of letters. Find all English words possible by taking one adjacent step to another letter, starting with any letter; one occurrence of each letter per word; you're given a dictionary (/usr/share/dict/words)

T	A	E
O	C	P



- For each node in graph, find all words
- For a specific starting node p, perform DFS; at each node, look up word consisting of all letters on path from p