Graphs

It’s all about relationships

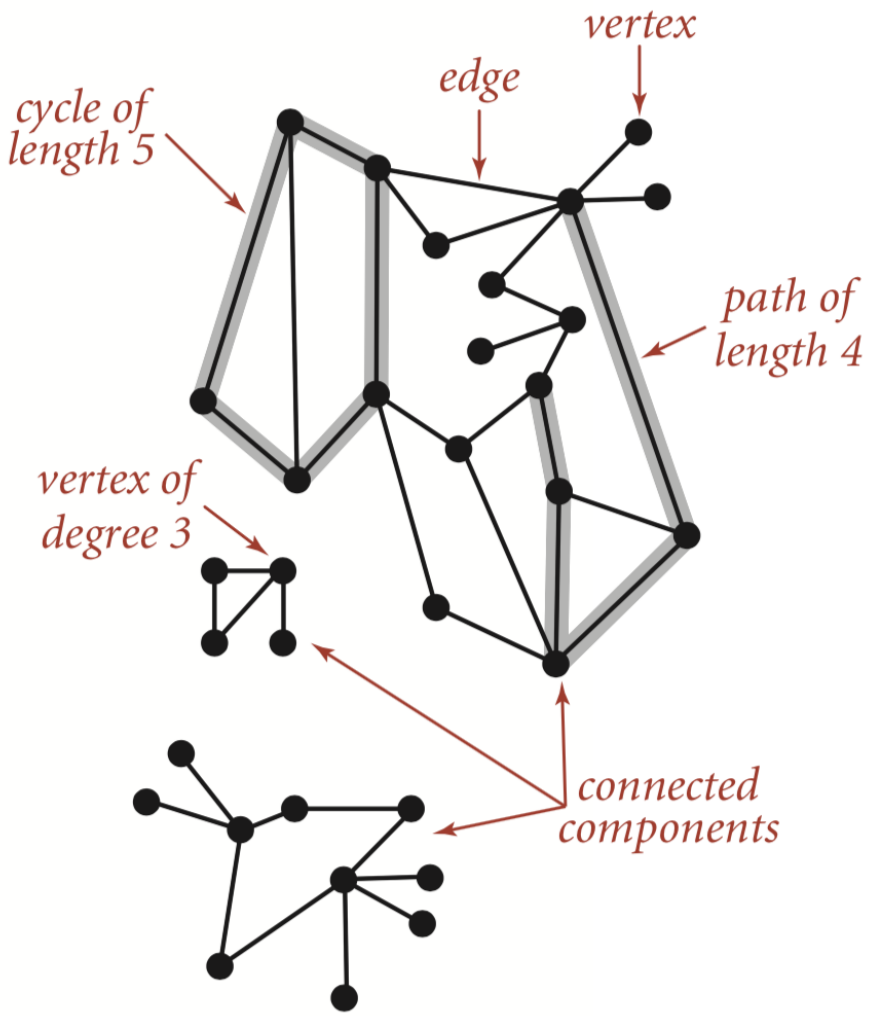
Mustafa Hajij
MSDS program
University of San Francisco

UNIVERSITY OF SAN FRANCISCO

What's a graph?

- A graph is a collection of connected element pairs, u→v or u−v
- As with a tree, a graph is the aggregate of nodes/edges
- Nodes can be email addresses, map locations, documents, tasks to perform, URLs on the web, customers, computers on 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 directed edges is >=0 and <= n^2 since n nodes could connect to all other nodes and itself; if no self cycles then the max is $n(n-1)$

Undirected graph, terms

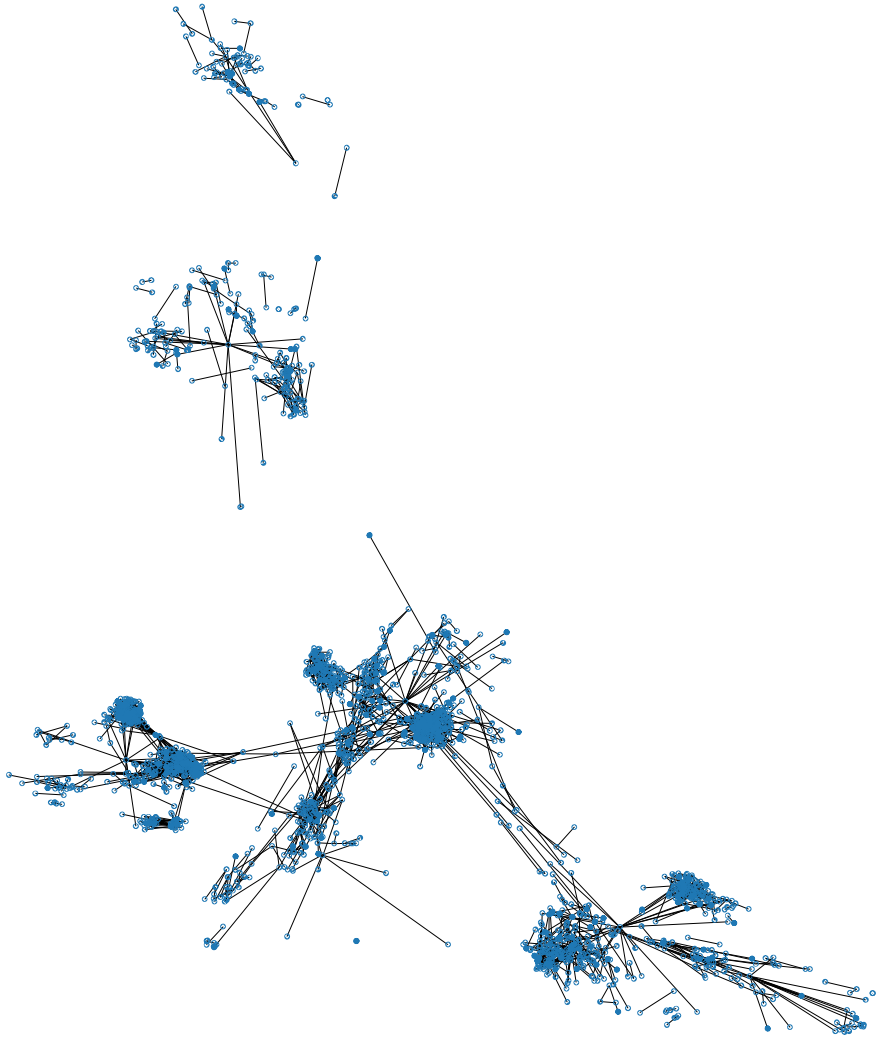


From: Algorithms book by Robert Sedgewick and Kevin Wayne

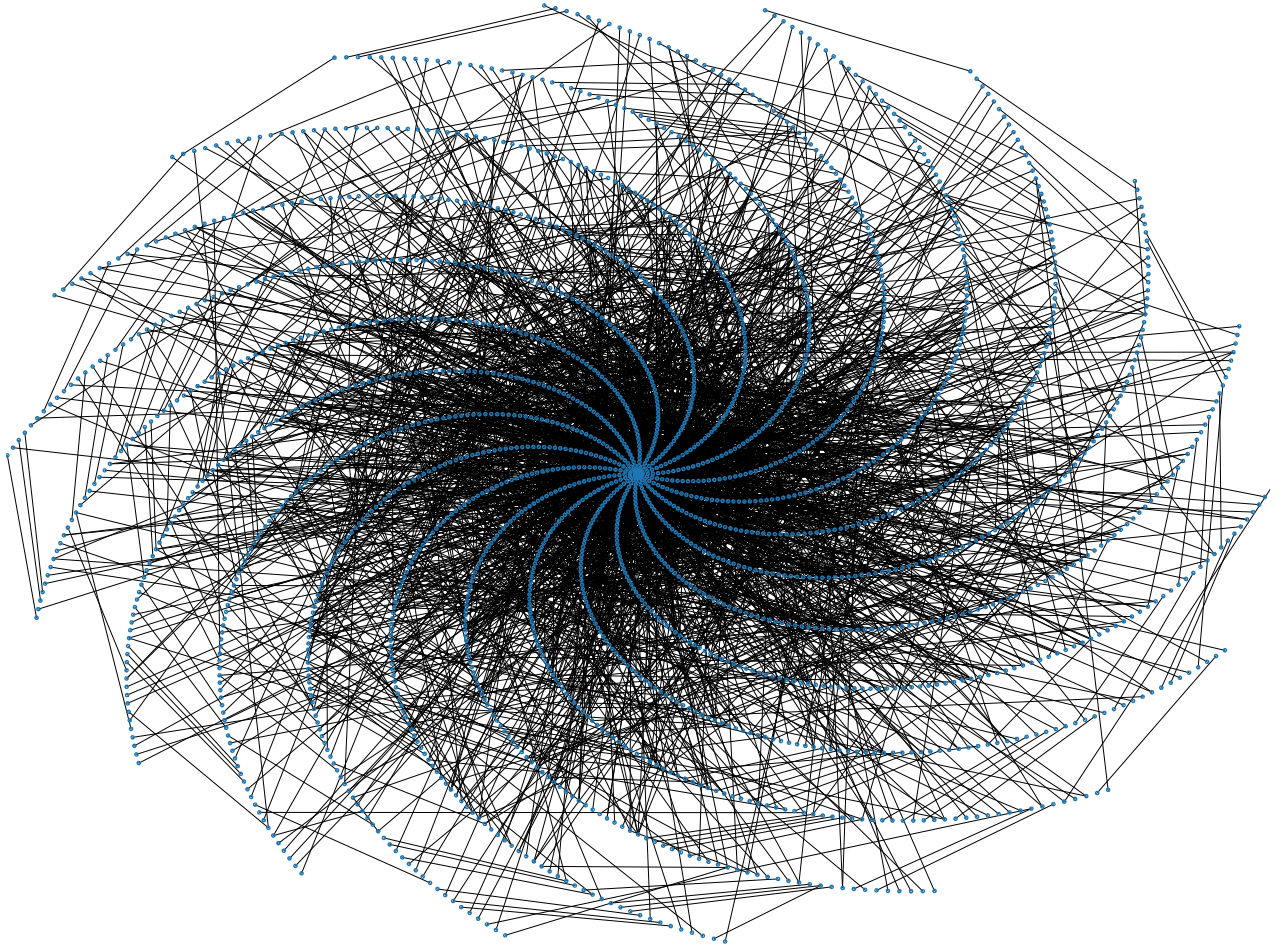
Uses networkx

Facebook friend network, different layouts

Fruchterman Reingold



spiral



UNIVERSITY OF SAN FRANCISCO

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]
  https://en.wikipedia.org/wiki/Travelling_salesman_problem

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 if we have unique node IDs

	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

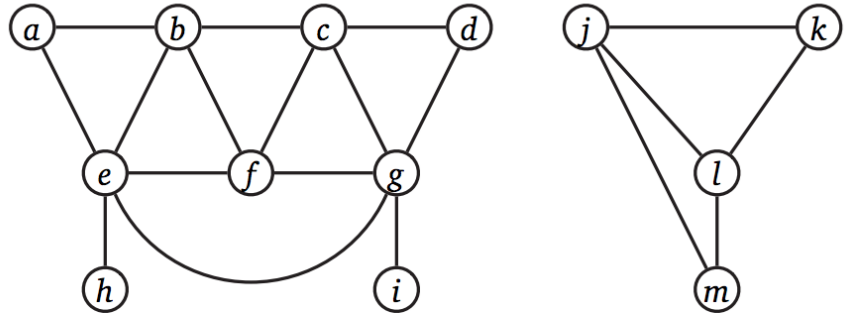


Figure 5.11. An adjacency matrix for our example graph.

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

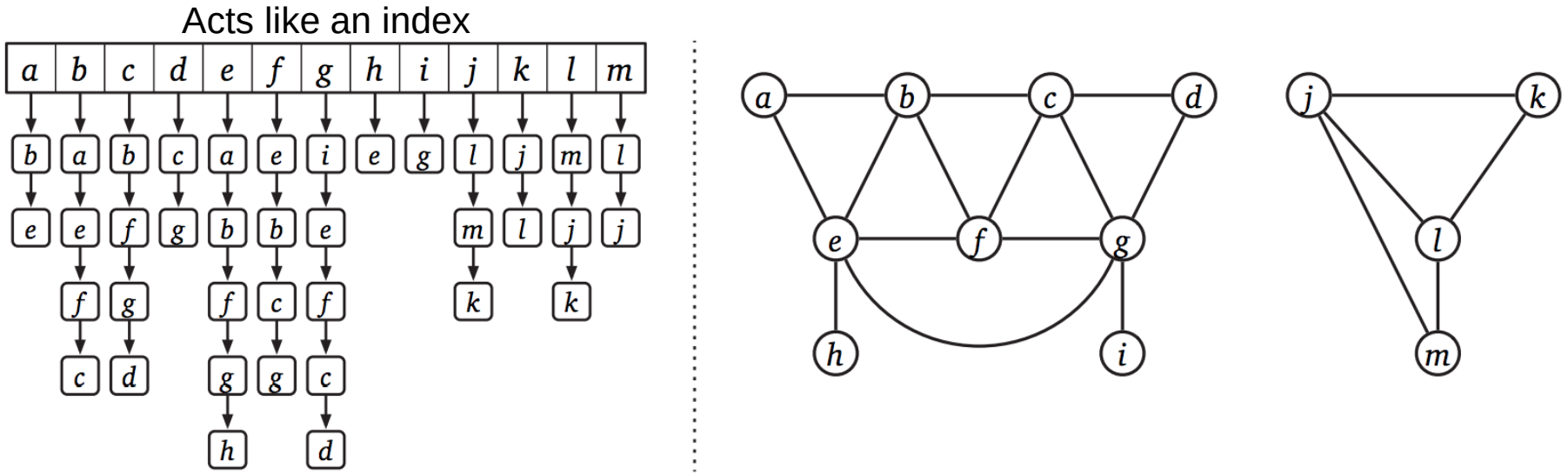


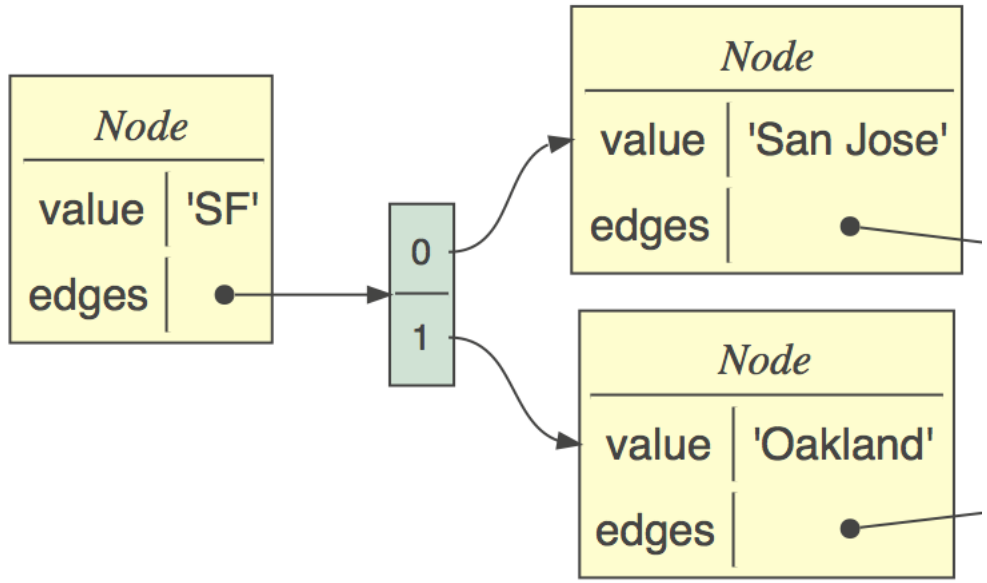
Figure 5.9. An adjacency list for our example graph.

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

Connected nodes implementation

- Common implementation due to nice mapping to objects
- Each node has info about that node and its 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 edge(self, target:Node):
    self.edges.append(target)
```

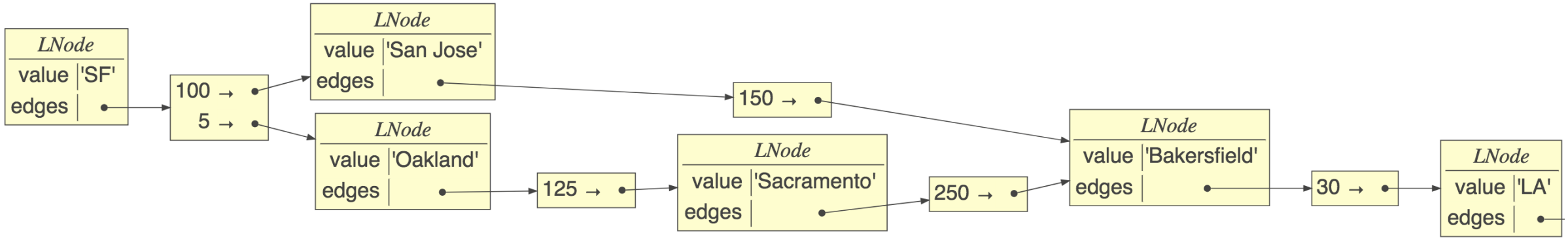


Implementation with labels

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

Edge->node is a dictionary, not list

```
sf.edge(100,sj)
sj.edge(150,baker)
...
```

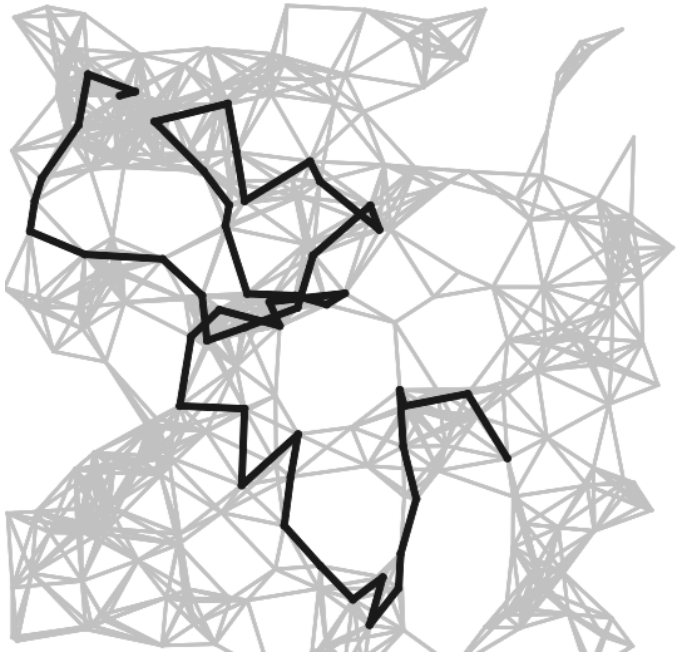


Depth-first search (review)

```
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
- Go deep first

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



Algorithms book by Sedgewick, Wayne

O(n,m) = n + m, for n **nodes**, m **edges**; m can be n^2
(counts node visits, edge traversals)

Is there a cycle (p can reach p)?

- E.g., can I get back to starting spot via one-way streets?
- If we run into starting node in visited set, return True; blue means different than plain walk:

```
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 don’t chase
    visited.add(p)
    for q in p.edges:
        c = iscyclic_(start, q, visited)
        if c: return True # found it, we can stop
    return False
```

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

Find set of nodes p can reach

- E.g., can we reach city y from city x (hint: Juneau not accessible)?
- Need two sets, one for avoiding cycles, another for reached nodes
- If we used visited for both, then p would also appear to reach itself, which might not be true; we add start node to visited but doesn't mean we can return to start via any path in graph

```
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) # add only if we traverse
        reachable_(q, reaches, visited)
```

Find set of nodes p can reach, track depth

- E.g., how many hops from person A to B in social network?
- Track node->depth map, not just set of nodes

```
def depths(p:Node) -> dict:
    reaches = {}
    depths_(p, reaches, depth=0)
    return reaches

def depths_(p:Node, reaches:dict, depth:int):
    if p in reaches: return
    reaches[p] = depth # distance to p from start (could be 0)
    for q in p.edges:
        depths_(q, reaches, depth+1)
```

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

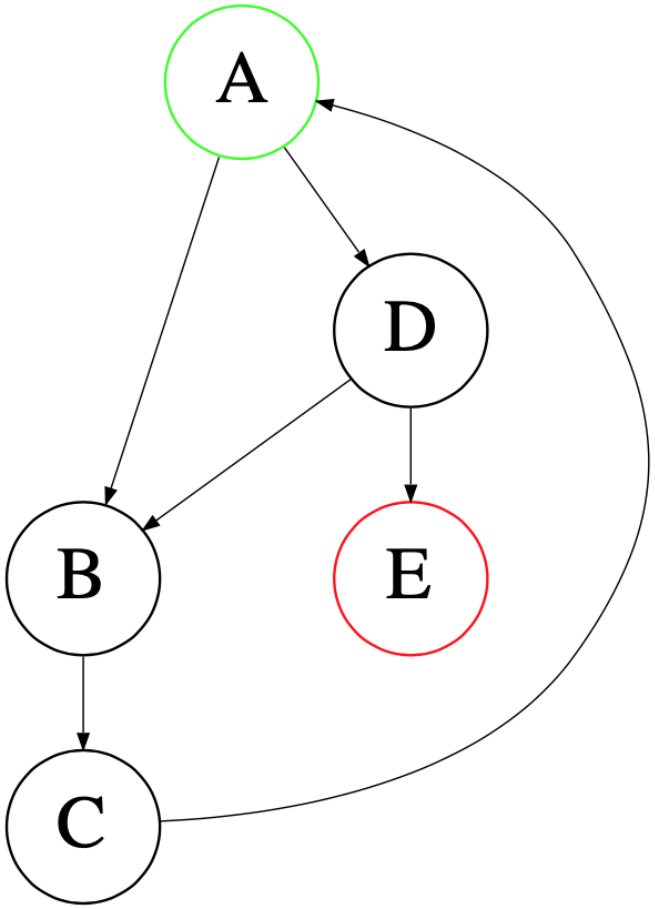
Find neighborhood within k edges

- E.g., find all friends within k friend hops
- Track dict node->depth, stop when we reach depth

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

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

Find first path from p to q



Track path to each node

path(A,E,[A])

path(B,E,[AB])

path(D,E,[AD])

path(C,E,[ABC])

path(B,E,[ADB])

path(E,E,[ADE])

path(A,E,[ABCA])

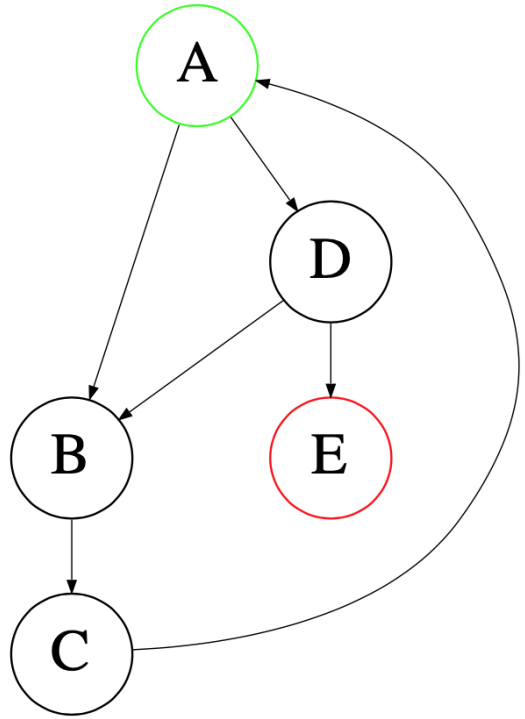
recursion tree

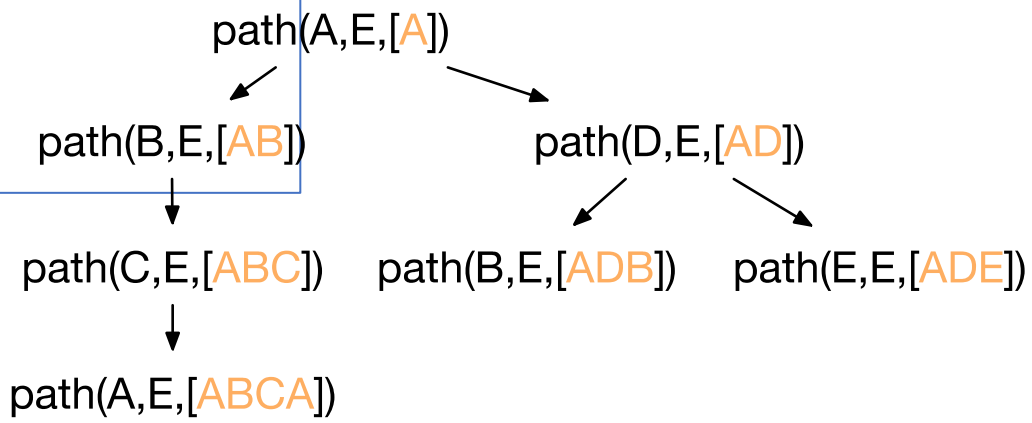
https://github.com/parrt/msds689/blob/master/notes/graphs.ipynb

Find first 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       # found q!
    if p in visited: return None  # avoid cycles
    visited.add(p)
    for e in p.edges:
        pa = path_(e, q, path+[e], visited)
        if pa is not None: return pa
    return None
```

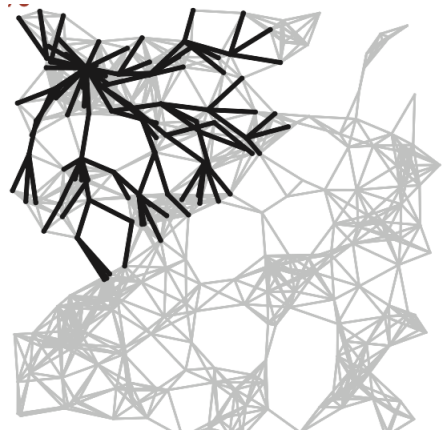




Must track path not just set of reachable nodes

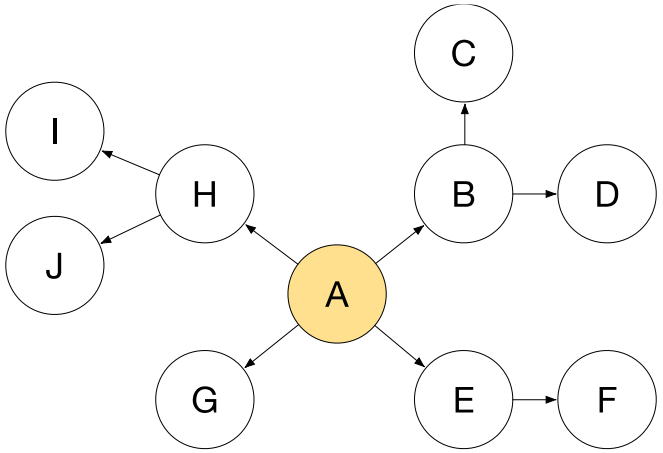
https://github.com/parrt/msds689/blob/master/notes/graphs.ipynb

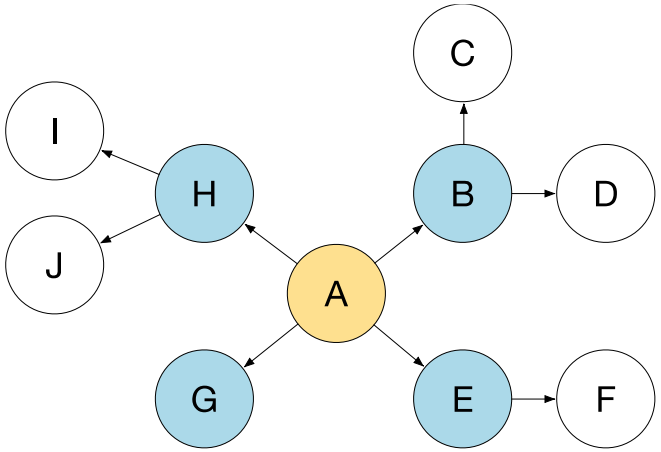
Breadth-first search vs DFS

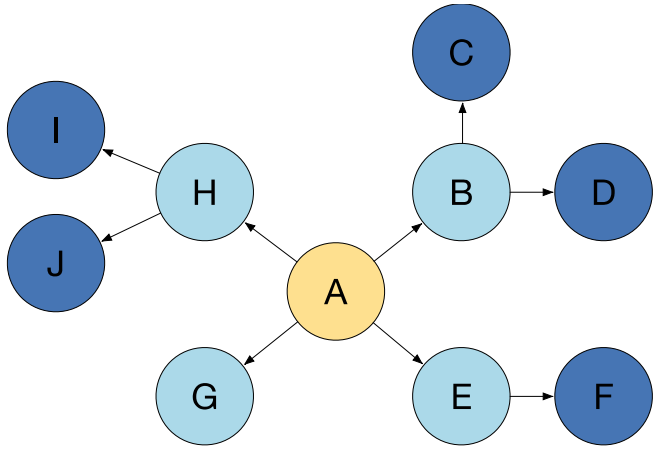


Algorithms book by Sedgewick, Wayne

- Visit all children then grandchildren…

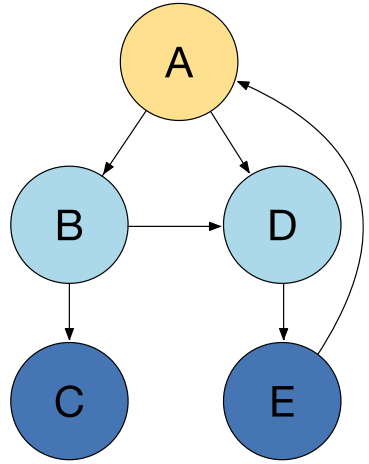






Todo: show worklist growing/shrinking here or next slide (animation)

BFS implementation



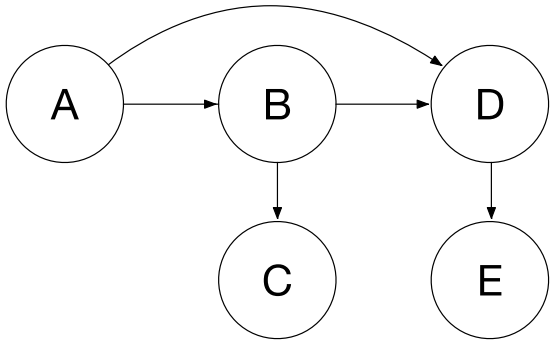
- Maintains work list of nodes and visited set
- **BFS** **DFS**

  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)
- *Nonrecursive*

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

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

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



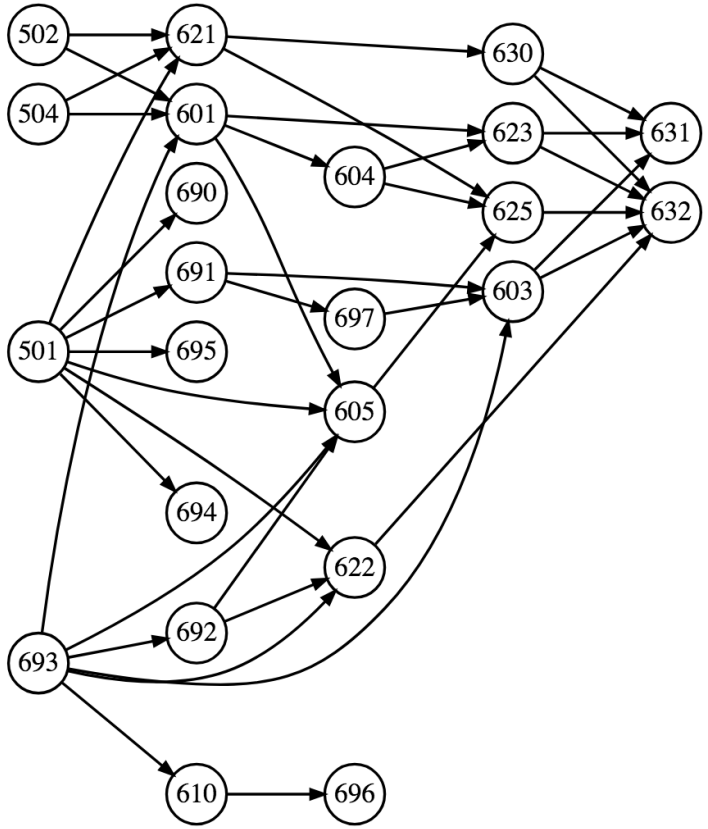
- BFS where work list is a list of paths not list of nodes
- Tail of path is where we left off working on it
- By searching all children before going deeper, we auto-magically find paths with shortest (unweighted) lengths

```
def shortest(root:Node, target:Node):
    visited = {root}
    worklist = [[root]] # list of paths
    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

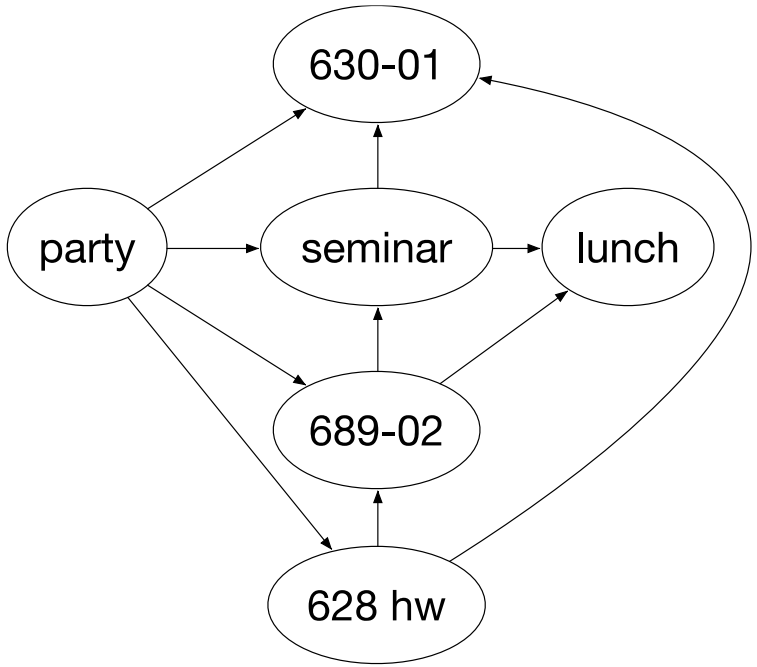
Topological sort (acyclic graphs)

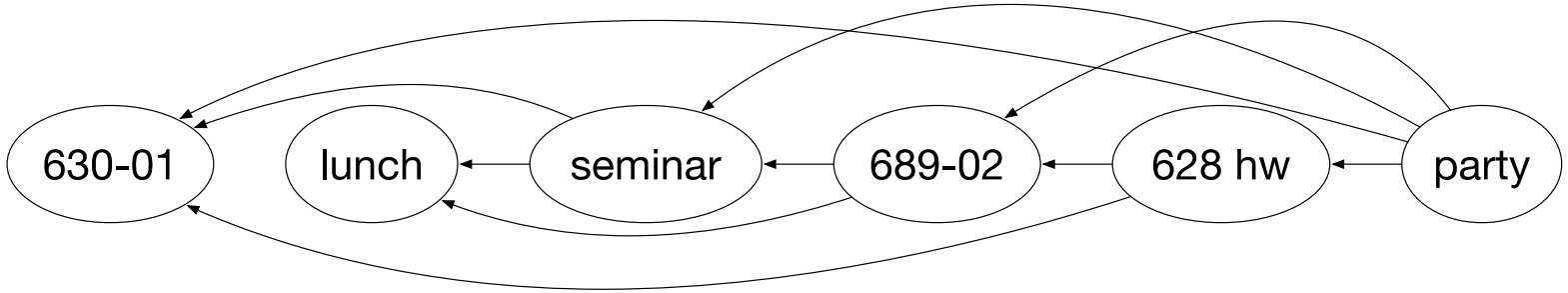
- **Problem**: Find linear ordering of nodes in directed acyclic graph such that all constraints, $u \rightarrow v$, are satisfied where u depends on v so v must come before u OR $u \rightarrow v$ mean u precedes v
- Examples: task ordering or course prereq chain.
- E.g., 502 is prereq for 621 and 601…
  Find order we should take classes
- Sort is not usually unique
- Cycles are meaningless for dependencies; how can 630-01 be attended before itself?



630-01

Example topological sort (u depends v)

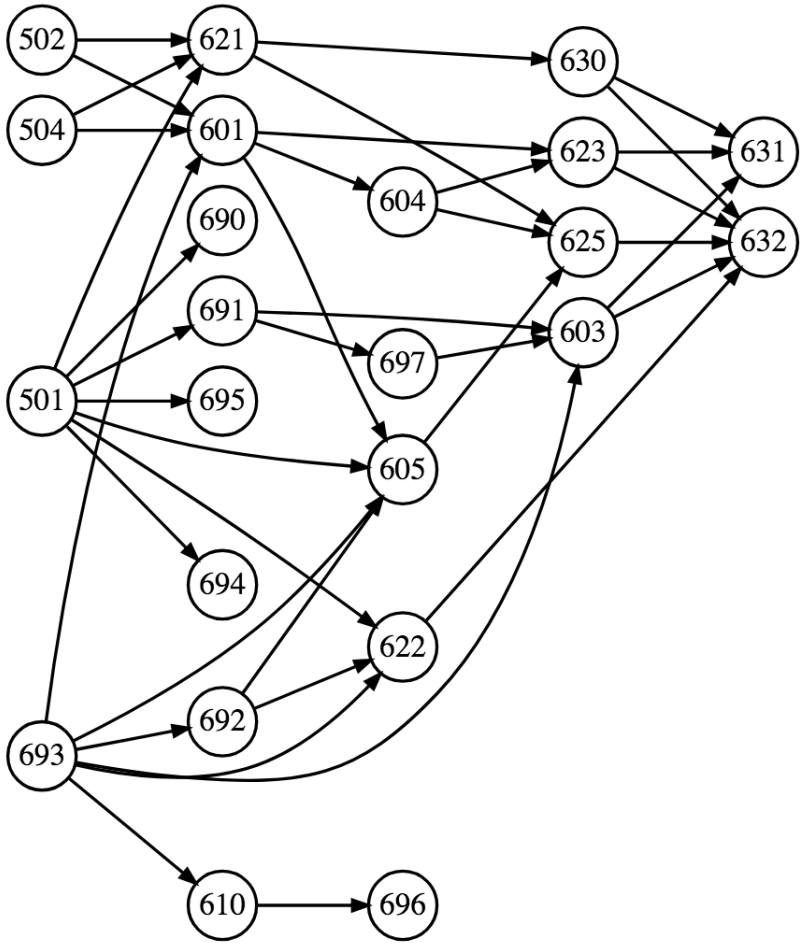


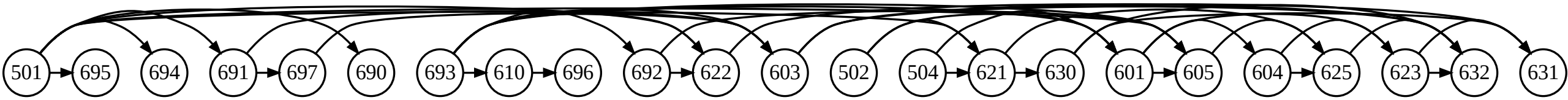


If u depends on v, any linear ordering where edges point to left is solution

Example where u precedes v

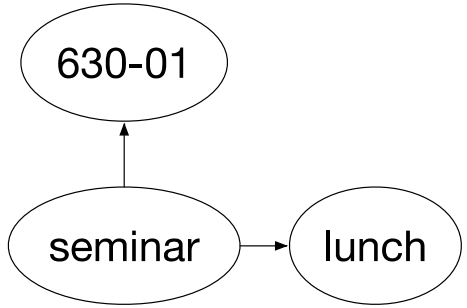
- In this case, edges must point to the right



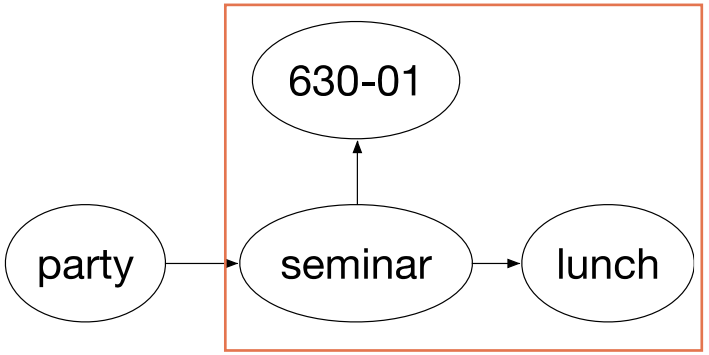


How to approach the problem

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



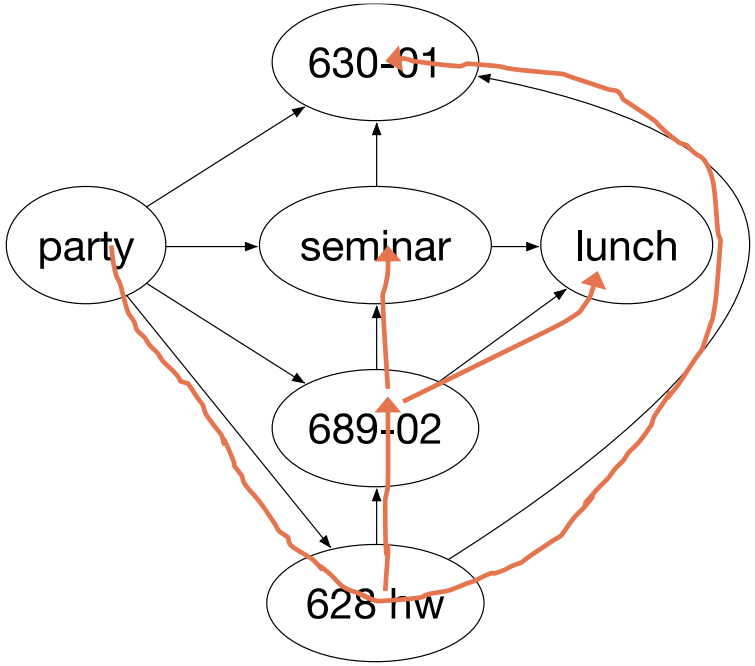
- What if we add party goal?
- "Perform all children tasks then yourself"



DFS-based topo sort implementation

- Lots of very complex algorithms on the web (not sure why)
- Simplest solution: DFS-based topological sort
- A valid sort is just the post-order graph traversal if u depends v!
- Perform all children tasks then yourself
- If u precedes v, reverse the result of post-order traversal;
  See proof page 582 of Sedgewick/Wayne Algorithms book
- 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/prints **after** visiting children:
  630-01, lunch, seminar, 689-02, 628, party
- Solution: 630-01, lunch, seminar, 689-02, 628 hw, party

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

DFS postorder traversal

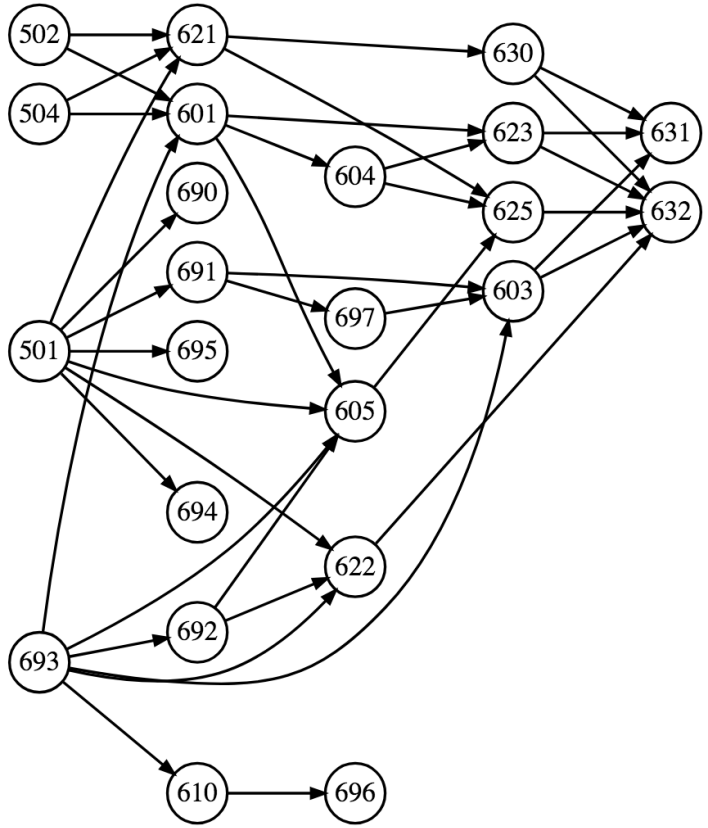
```
def postorder(p:Node, sorted:list, visited:set):
    if p in visited: return
    visited.add(p)
    for q in p.edges:
        postorder(q, sorted, visited)
    sorted.append(p) # postorder done after kids
```

(recall: no cycles)

With multiple roots, hit them all

```
MSDS = """501->690
502,504,693->601
601->604
601,604->623
..."""
```

```
# nodes is dict edge list mapping from->to
def toposort(nodes:dict):
    sorted = []
    visited = set()
    while len(visited) < len(nodes):
        todo = [node for node in nodes.values()
                if node not in visited]
        if len(todo)>0:
            anode = todo[0] # pick a node
            postorder(anode, sorted, visited)
    return reverse(sorted)
```



We reverse postorder here since u precedes v

Summary

- Graphs are for showing relationships between elements
- DFS for finding a path or multiple paths or cycles (recursive backtracking to find all nodes)
- BFS for find shortest (in edges) path or neighborhood
- DFS postorder great for topo sort
- Recursive alg's all use **visited** set or similar 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

Exercise

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

Exercise

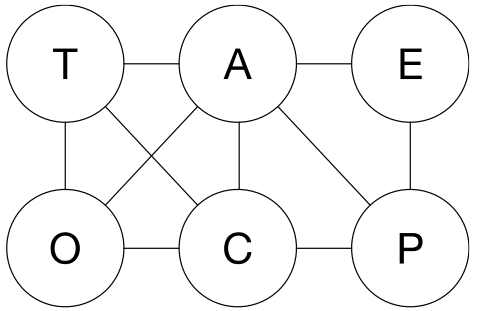
- Given 2 lists P,Q and function conn(p,q)=true if edge p→q exists. P is origin (starting) nodes and Q destination nodes. Report True for P[i] reaches Q[i] directly or indirectly
- E.g., P, Q could be cities and conn(p,q)=true if direct flight p→q
- Create graph using conn(p,q) for all nodes in P and 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

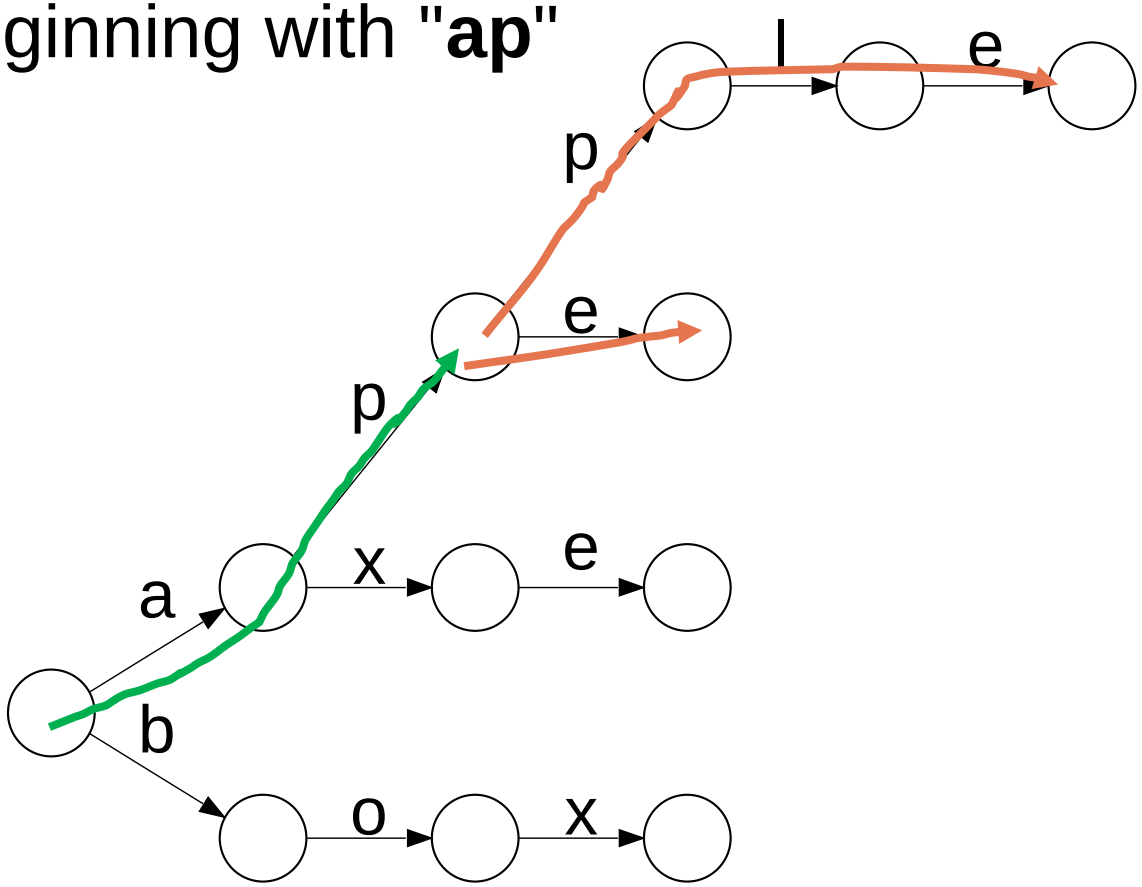
O C P



- For each node in graph, find all words
- For a specific starting node p, perform cycle-avoiding DFS; at each node q, look up word consisting of all letters on path from p to q, checking for duplicated letters

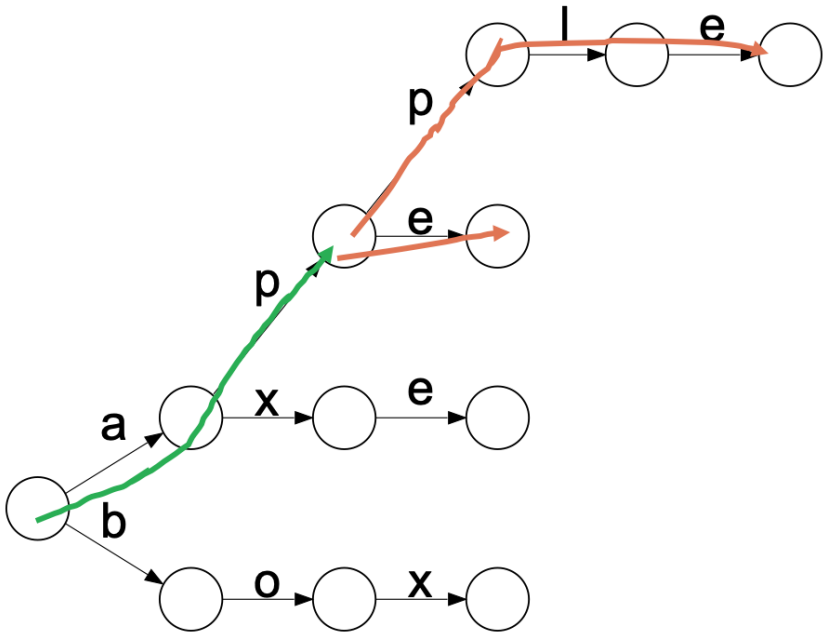
Exercise: *Find all words with given prefix*

- Find words beginning with "**ap**"



See https://github.com/parrt/msds689/blob/master/notes/code/prefix_trie.py

Exercise: *Find all words with given prefix*

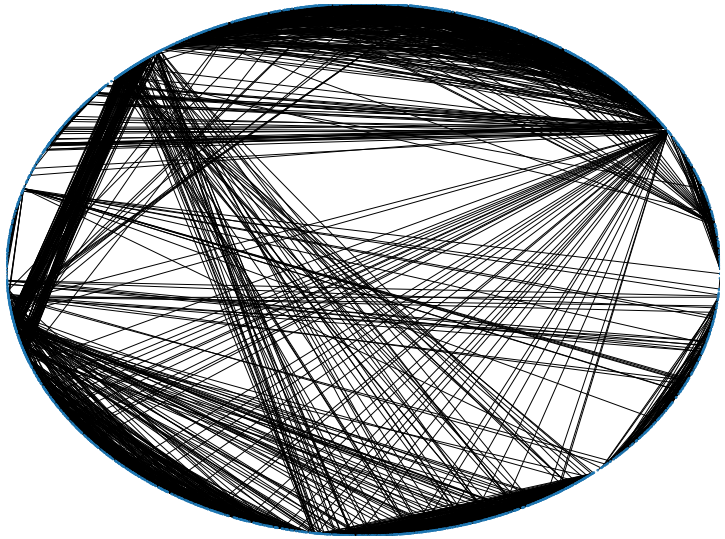


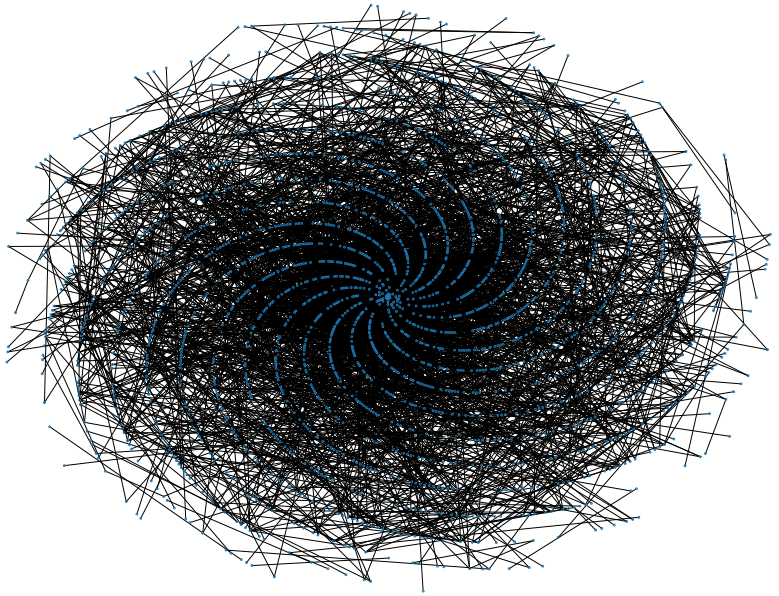
- Build a Trie with dictionary words
- Find node in trie reached by prefix; call it p (green line)
- Recursively, using depth first search, find all reachable leaf (stop/accept) nodes (orange lines)
- Pass a path string as arg down the recursion chain to incrementally build strings reachable from p
  `def suffixes_(p:TrieNode, path, paths): …`
- There can be no cycle so we don't worry about "visited" arg, just track the current path and the overall list of paths we found

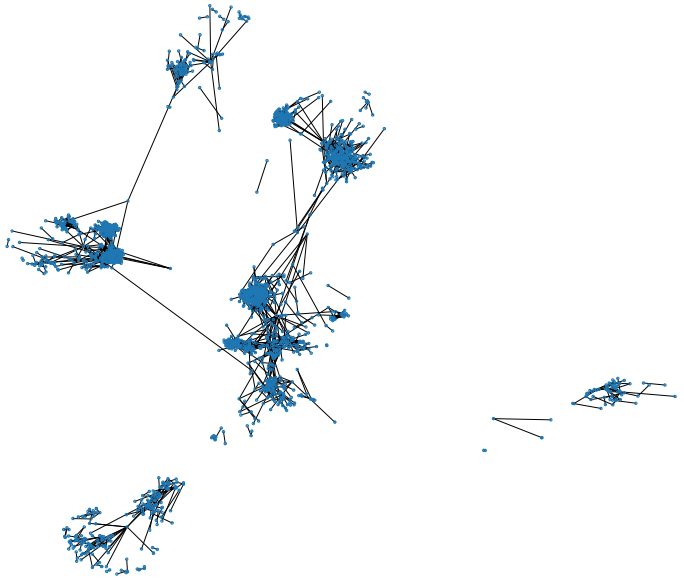
See https://github.com/parrt/msds689/blob/master/notes/code/prefix_trie.py

Coding exercise:
Visualize: FB friend graph with networkx

- pip install networkx
- download and uncompress *https://snap.stanford.edu/data/facebook_combined.txt.gz*
- Get small sample of edges then do edge_subgraph(), draw_networkx_edges(), draw_networkx_nodes()
- Need to pass positions of nodes for layout, such as circular_layout()



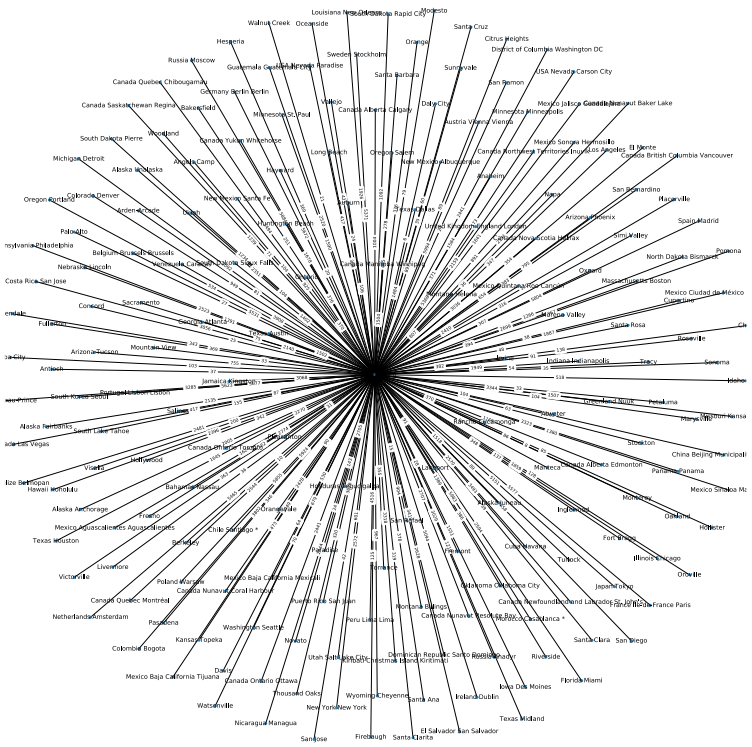


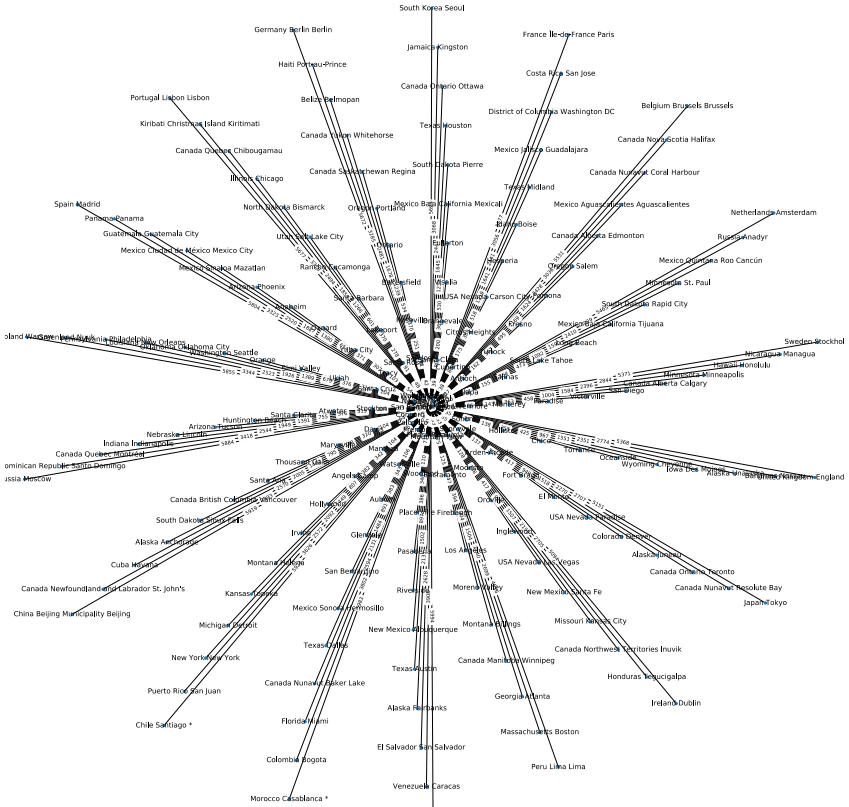


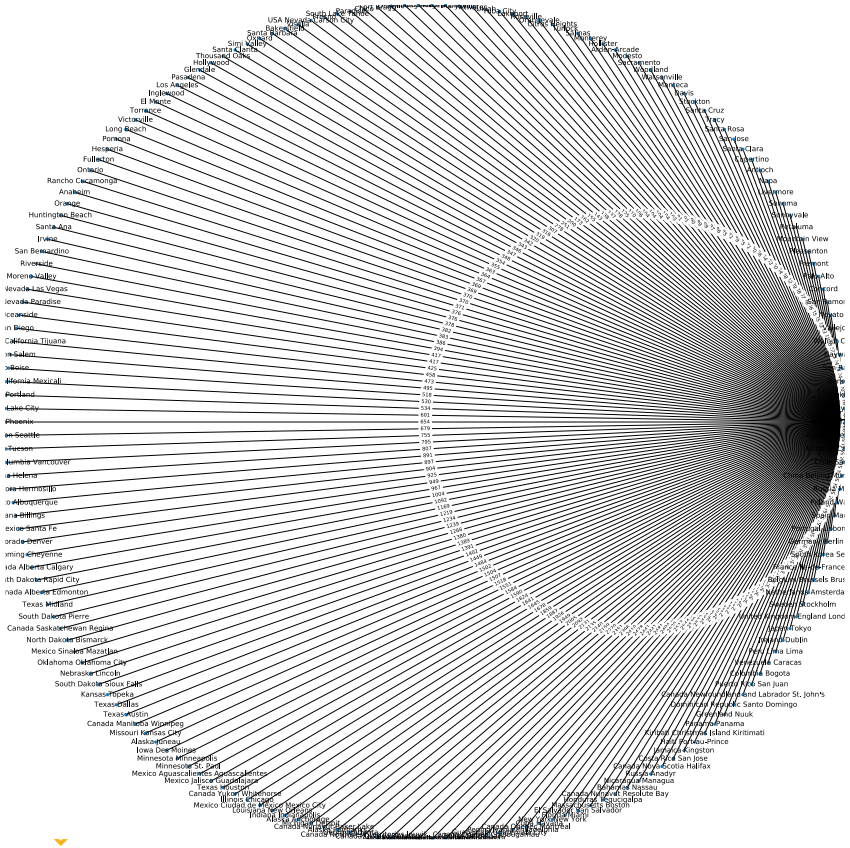
See https://github.com/parrt/msds689/blob/master/notes/code/viz_facebook.py

Coding exercise
networkx: distances from SF to other cities

- Download data: https://github.com/parrt/msds689/blob/master/notes/code/distances.csv
- Create graph showing SF to other cities using multiple layouts







See https://github.com/parrt/msds689/blob/master/notes/code/sfo_city_distances.py