BFS and DFS applications

Tyler Moore

CSE 3353, SMU, Dallas, TX

Lecture 7

Some slides created by or adapted from Dr. Kevin Wayne. For more information see

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

BFS/DFS Applications

- Shortest path between two nodes in a graph
- Topological sorting
- Finding connected components

Connectivity

s-t connectivity problem. Given two node s and t, is there a path between s and t?

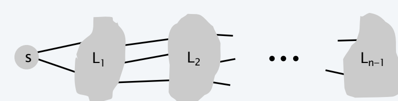
s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

Applications.

- Friendster.
- Maze traversal.
- Kevin Bacon number.
- Fewest number of hops in a communication network.

Breadth-first search

BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time.



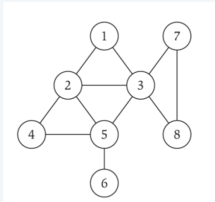
BFS algorithm.

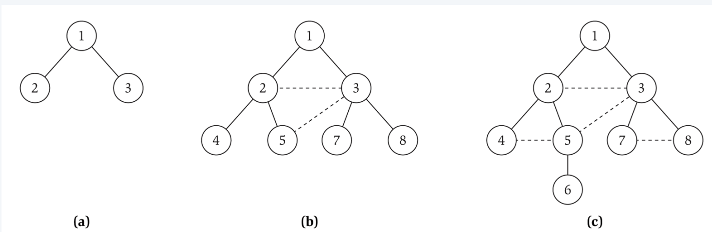
- $L_0 = \{ s \}$.
- L_1 = all neighbors of L_0.
- L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
- L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.

Theorem. For each i, L_i consists of all nodes at distance exactly i from s. There is a path from s to t iff t appears in some layer.

Breadth-first search

Property. Let T be a BFS tree of $G = (V, E)$, and let (x, y) be an edge of G. Then, the level of x and y differ by at most 1.





Breadth-first search: analysis

Theorem. The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

Pf.

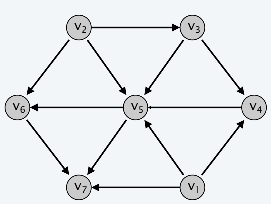
- Easy to prove $O(n^2)$ running time:
  - at most n lists $L[i]$
  - each node occurs on at most one list; for loop runs $\leq n$ times
  - when we consider node u, there are $\leq n$ incident edges (u, v), and we spend $O(1)$ processing each edge
- Actually runs in $O(m + n)$ time:
  - when we consider node u, there are $degree(u)$ incident edges (u, v)
  - total time processing edges is $\Sigma_{u \in V}\ degree(u) = 2m$. ▪

each edge (u, v) is counted exactly twice in sum: once in degree(u) and once in degree(v)

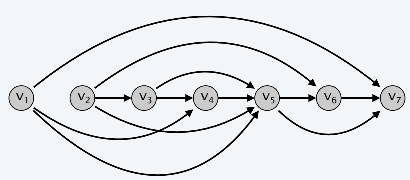
Directed acyclic graphs

Def. A DAG is a directed graph that contains no directed cycles.

Def. A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.



a DAG



a topological ordering

Application of topological sorting

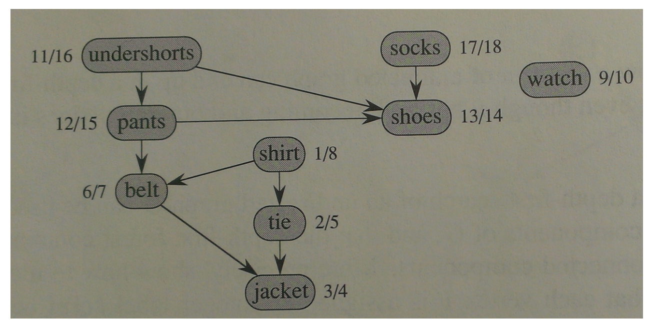


Figure : Directed acyclic graph for clothing dependencies

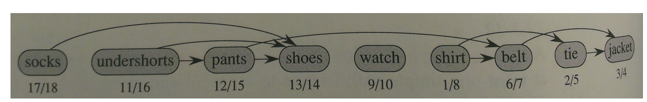


Figure : Topological sort of clothes

Precedence constraints

Precedence constraints. Edge (v_i, v_j) means task v_i must occur before v_j.

Applications.

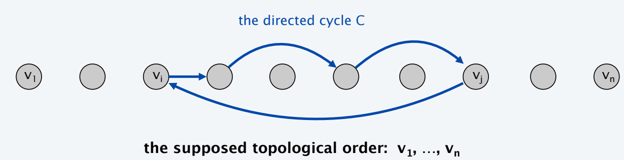
- Course prerequisite graph: course v_i must be taken before v_j.
- Compilation: module v_i must be compiled before v_j. Pipeline of computing jobs: output of job v_i needed to determine input of job v_j.

Directed acyclic graphs

Lemma. If G has a topological order, then G is a DAG.

Pf. [by contradiction]

- Suppose that G has a topological order $v_1, v_2, \ldots, v_n$ and that G also has a directed cycle C. Let's see what happens.
- Let v_i be the lowest-indexed node in C, and let v_j be the node just before v_i; thus (v_j, v_i) is an edge.
- By our choice of i, we have $i < j$.
- On the other hand, since (v_j, v_i) is an edge and $v_1, v_2, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction. ▪



Directed acyclic graphs

Lemma. If G has a topological order, then G is a DAG.

Q. Does every DAG have a topological ordering?

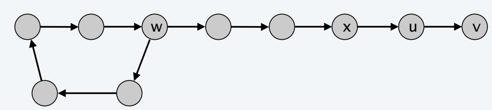
Q. If so, how do we compute one?

Directed acyclic graphs

Lemma. If G is a DAG, then G has a node with no entering edges.

Pf. [by contradiction]

- Suppose that G is a DAG and every node has at least one entering edge. Let's see what happens.
- Pick any node v, and begin following edges backward from v. Since v has at least one entering edge (u, v) we can walk backward to u.
- Then, since u has at least one entering edge (x, u), we can walk backward to x.
- Repeat until we visit a node, say w, twice.
- Let C denote the sequence of nodes encountered between successive visits to w. C is a cycle. ▪



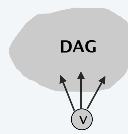
Directed acyclic graphs

Lemma. If G is a DAG, then G has a topological ordering.

Pf. [by induction on n]

- Base case: true if $n = 1$.
- Given DAG on $n > 1$ nodes, find a node v with no entering edges.
- $G - \{ v \}$ is a DAG, since deleting v cannot create cycles.
- By inductive hypothesis, $G - \{ v \}$ has a topological ordering.
- Place v first in topological ordering; then append nodes of $G - \{ v \}$
- in topological order. This is valid since v has no entering edges. ▪

```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
  and append this order after v
```



50

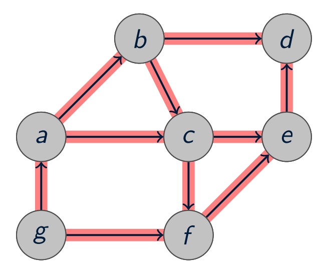
Examples of Induction-Based Topological Sorting

Python code for induction-based topsort

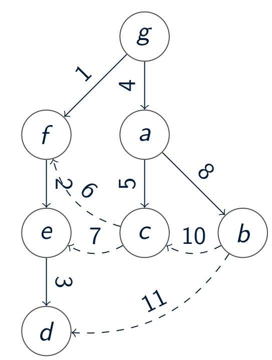
```
def topsort(G):
    count = dict((u, 0) for u in G)#The in-degree for each node
    for u in G:
        for v in G[u]:
            count[v] += 1      #Count every in-edge
    Q = [u for u in G if count[u] == 0] # Valid initial nodes
    S = []                     #The result
    while Q:                   #While we have start nodes...
        u = Q.pop()            #Pick one
        S.append(u)            #Use it as first of the rest
        for v in G[u]:
            count[v] -= 1      #"Uncount" its out-edges
            if count[v] == 0:#New valid start nodes?
                Q.append(v)    #Deal with them next
    return S
```


Topological sorting on DAGs

Directed Acyclic Graph



Depth-First Search Tree

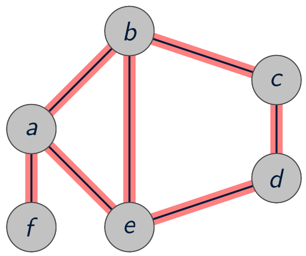


Discovered: g f e d a c b
Processed: d e f c b a g

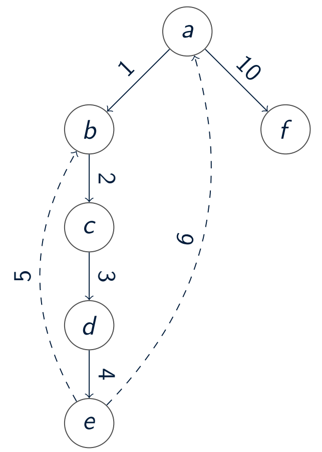
Topological sort: g a b c f e d

DFS Trees: all descendants of a node u are processed after u is discovered but before u is processed

Undirected Graph



Depth-First Search Tree



Discovered: a b c d e f
Processed: e d c b f a

How can we tell if one node is a descendant of another?

- Answer: with depth-first timestamps!
- After we create a graph in a depth-first traversal, it would be nice to be able to verify if node A is encountered before node B, etc.
- We add one timestamp for when a node is discovered (during preorder processing) and another timestamp for when a node is processed (during postorder processing)

Code for depth-first timestamps

```
def dfs(G, s, d, f, S=None, t=0):
    if S is None: S = set()# Initialize the history
    d[s] = t; t += 1          # Set discover time
    S.add(s)                  # We've visited s
    for u in G[s]:            # Explore neighbors
        if u in S: continue# Already visited. Skip
        t = dfs(G, u, d, f, S, t)                    # Recurse; update timestamp
    f[s] = t; t += 1          # Set finish time
    return t                  # Return timestamp
```

```
>>> f={}
>>> d={}
>>> dfs(N,'a',d,f)
12
>>> d
{'a': 0, 'c': 2, 'b': 1, 'e': 4, 'd': 3, 'f': 9}
>>> f
{'a': 11, 'c': 7, 'b': 8, 'e': 5, 'd': 6, 'f': 10}
```

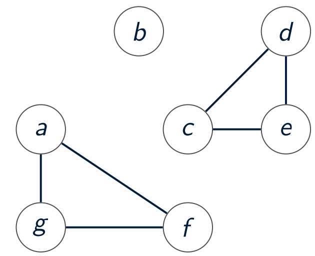

Using depth-first timestamps for topological sorting

```
>>> f={}
>>> d={}
>>> dfs(DAG,'g',d,f)
14
>>> topsort = [k for k,v in sorted(f.iteritems(),
                                    key=lambda(k,v): v)]
>>> topsort.reverse()
>>> topsort
['g', 'a', 'b', 'c', 'f', 'e', 'd']
```

Exercise: DFS-Based Topological Sorting

Connected Components

- A connected component of an undirected graph is a maximal set of vertices such that there is a path between every pair of vertices



- **Exercise**: Explain in English how you could find all connected components of a graph using breadth-first search.

Code to find connected components

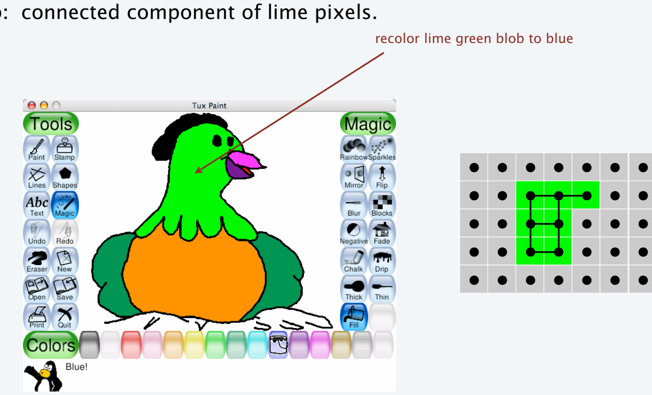
```
def find_components(G):
    vertices = G.keys()
    u = vertices[0]           #pick starting vertex
    components=[]             #list of components
    S =set()                  #discovered vertices
    while True:
        cc = list(bfs(G,u))  #do BFS from vertex
        S.update(cc)          #update discovered
        components.append(cc)#update component list
        for v in cc:          #remove component's vertices
            vertices.remove(v)#from set to check
        if not vertices: break
        u = vertices[0]       #pick the next undiscovered vertex
    return components

>>> find_components(G)
[['a', 'g', 'f'], ['c', 'e', 'd'], ['b']]
```


Flood fill

Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.



22

Flood fill

Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.

recolor lime green blob to blue