Introduction to Graphs

Tyler Moore

CS 2123, The University of Tulsa

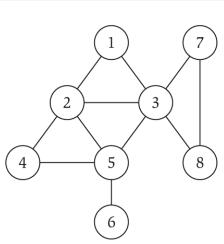
Some slides created by or adapted from Dr. Kevin Wayne. For more information see

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

Undirected graphs

Notation. $G = (V, E)$

- V = nodes.
- E = edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: $n = |V|, m = |E|$.



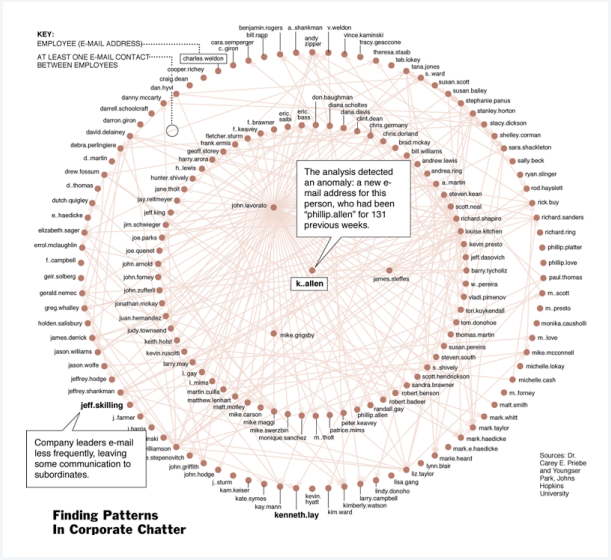
$V = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}$

$E = \{ 1\text{-}2, 1\text{-}3, 2\text{-}3, 2\text{-}4, 2\text{-}5, 3\text{-}5, 3\text{-}7, 3\text{-}8, 4\text{-}5, 5\text{-}6, 7\text{-}8 \}$

$m = 11, n = 8$

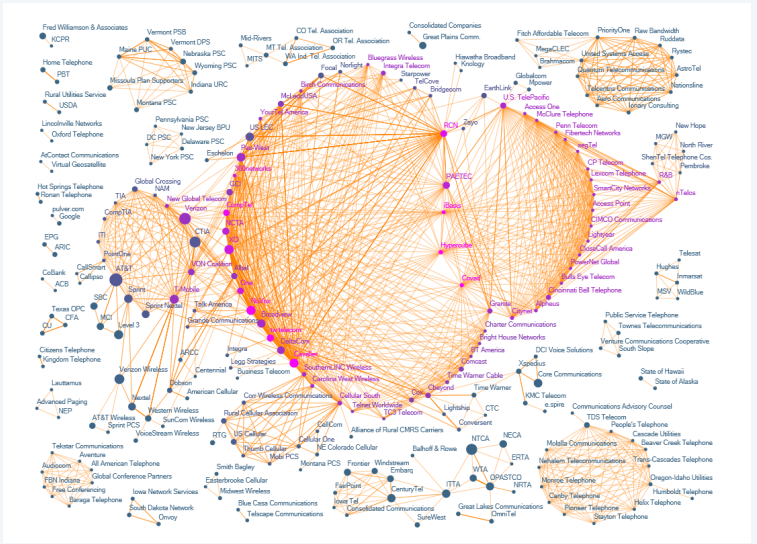
3

One week of Enron emails



4

The evolution of FCC lobbying coalitions



"The Evolution of FCC Lobbying Coalitions" by Pierre de Vries in JoSS Visualization Symposium 2010

5

Framingham heart study

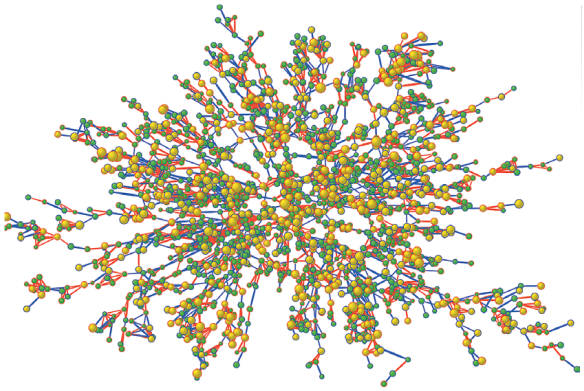
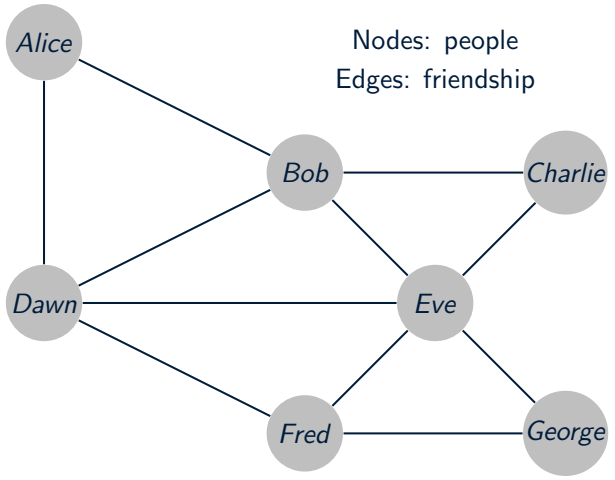


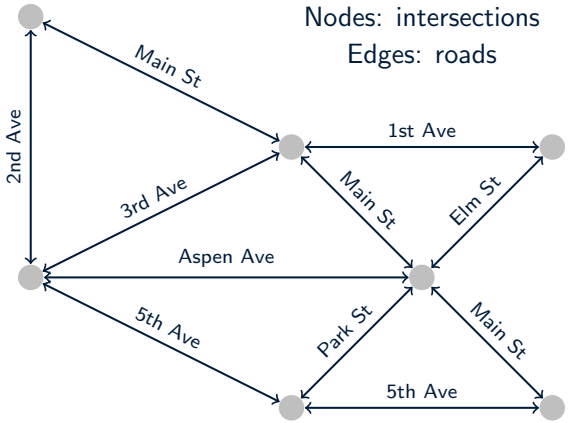
Figure 1. Largest Connected Subcomponent of the Social Network in the Framingham Heart Study in the Year 2000.
Each circle (node) represents one person in the data set. There are 2200 persons in this subcomponent of the social network. Circles with red borders denote women, and circles with blue borders denote men. The size of each circle is proportional to the person's body-mass index. The interior color of the circles indicates the person's obesity status: yellow denotes an obese person (body-mass index, ≥30) and green denotes a nonobese person. The colors of the ties between the nodes indicate the relationship between them: purple denotes a friendship or marital tie and orange denotes a familial tie.

"The Spread of Obesity in a Large Social Network over 32 Years" by Christakis and Fowler in New England Journal of Medicine, 2007 6

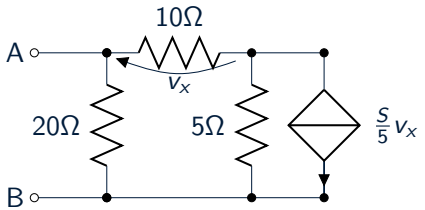
Social Network as a Graph



Road Network as a Graph

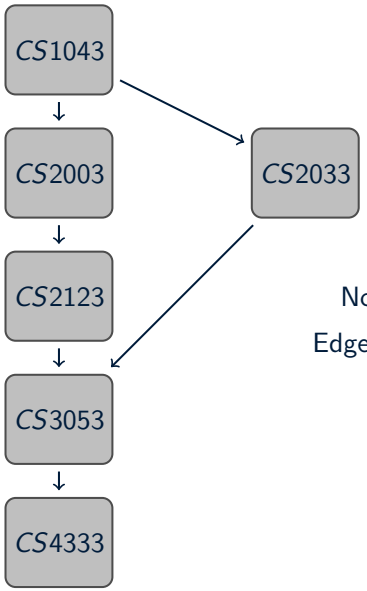


Electronic Circuits as a Graph



- Vertices: junctions
- Edges: components

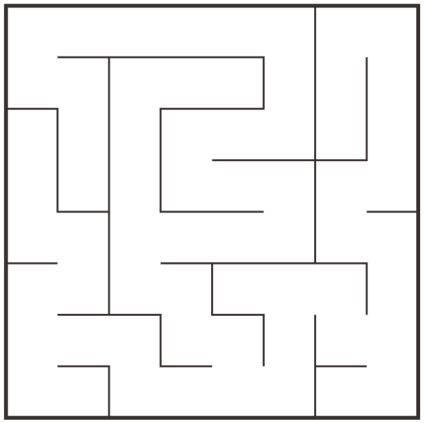
Course Dependencies as a Graph

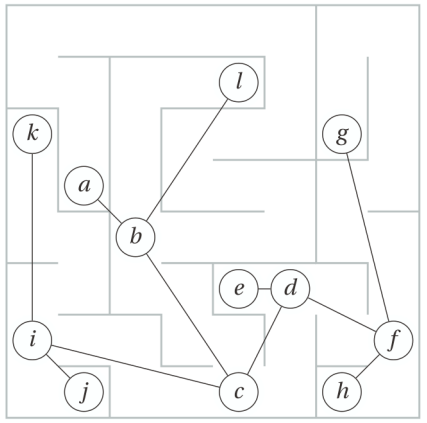


Nodes: courses

Edges: prerequisites

Maze as a Graph



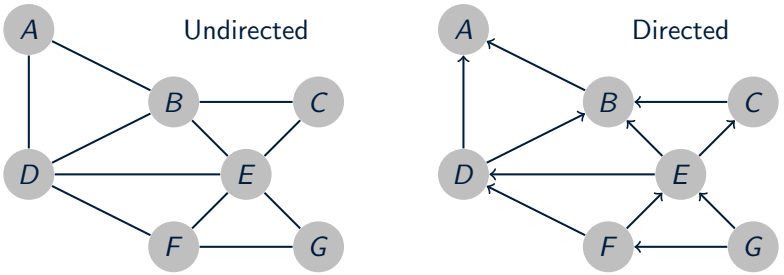


Nodes: rooms; edges: doorways

Flavors of Graphs

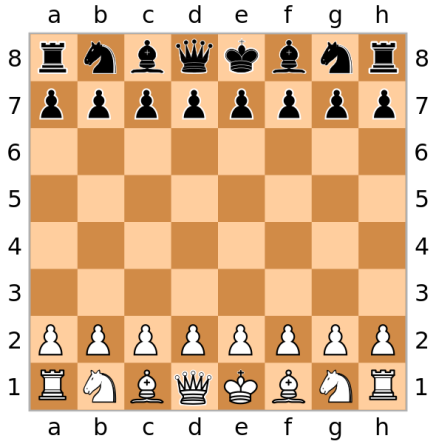
- The first step in any graph problem is recognizing that you have a graph problem
- It's not always obvious!
- The second step in any graph problem is determining which flavor of graph you are dealing with
- Learning to talk the talk is an important part of walking the walk
- The flavor of graph has a big impact on which algorithms are appropriate and efficient

Directed vs. Undirected Graphs



- A graph $G = (V, E)$ is undirected if edge $(x, y) \in E$ implies that (y, x) is also in E
- Road networks between cities are typically undirected
- Street networks within cities are almost always directed because of one-way streets
- What about online social networks?

Exercise 1: Spot the graph



- Nodes:
- Edges:

Exercise 2: Spot the graph

Pay From	Pay To	Amount	Date
12354	67324	$1000	Sep 1
12398	67324	$500	Sep 4
67324	45721	$750	Sep 7
78923	12398	$500	Sep 8

- Nodes:
- Edges:

Exercise 3: Spot the graph

```
x = ['ham','spam','glam','tram']
for a in x:
    print a
    if a =='ham':
       print 'awesome'
    else: print 'terrible'
    print 'moving on'
```

- Nodes:
- Edges:

Exercise 4: Spot the graph

```
def foo(arg1):
    x = arg1+2
    bar(x)

def bar(arg1,arg2):
    y = arg1+arg2
    ham(y)

def ham(arg1):
    bar(2*arg1)
```

- Nodes:
- Edges:

Some graph applications

graph	node	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position	legal move
social relationship	person, actor	friendship, movie cast
neural network	neuron	synapse
protein network	protein	protein-protein interaction
molecule	atom	bond

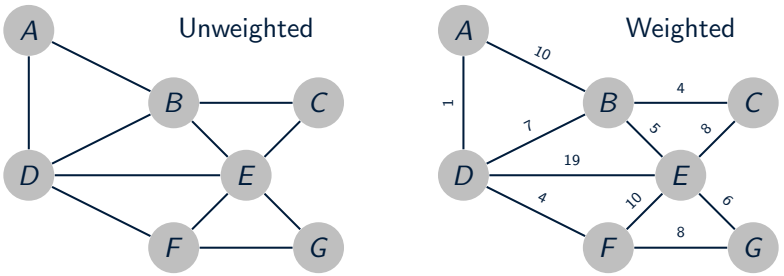
7

Some directed graph applications

directed graph	node	directed edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
WordNet	synset	hypernym
scheduling	task	precedence constraint
financial	bank	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

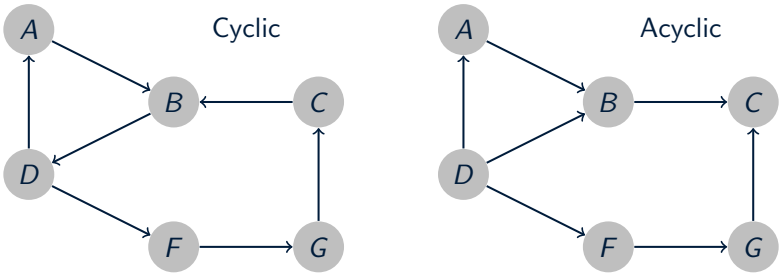
39

Weighted vs. Unweighted Graphs



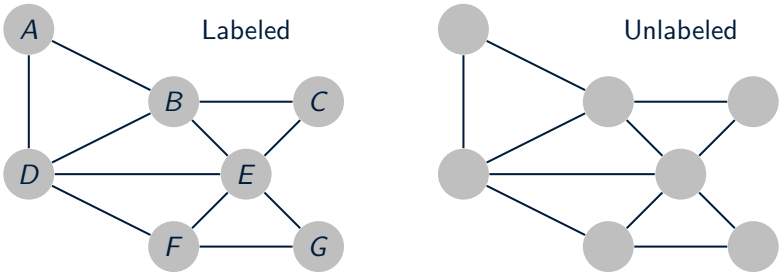
- In weighted graphs, each edge (or vertex) of G is assigned a numerical value, or weight
- The edges of a road network graph might be weighted with their length, drive-time or speed limit
- In unweighted graphs, there is no cost distinction between various edges and vertices

Cyclic vs. Acyclic Graphs



- An acyclic graph does not contain any cycles
- Trees are connected, acyclic, undirected graphs
- Directed acyclic graphs are called DAGs
- DAGs arise naturally in scheduling problems, where a directed edge (x, y) indicates that x must occur before y.

Labeled vs. Unlabeled Graphs



- In labeled graphs, each vertex is assigned a unique name or identifier to distinguish it from all other vertices.
- An important graph problem is isomorphism testing: determining whether the topological structure of two graphs are in fact identical if we ignore any labels.

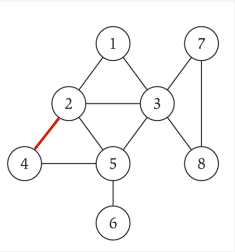
More Graph Terminology

- A path is a sequence of edges connecting two vertices
- Two common problems: (1) does a path exist between A and B? (2) what is the shortest path between A and B?
- A graph is connected if there is a path between any two vertices
- A directed graph is strongly connected if there is a directed path between any two vertices.
- The degree of a vertex is the number of edges adjacent to it

Graph representation: adjacency matrix

Adjacency matrix. n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge.

- Two representations of each edge.
- Space proportional to n^2.
- Checking if (u, v) is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.



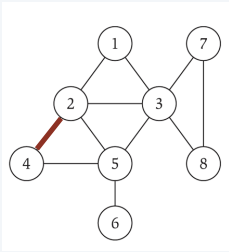
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

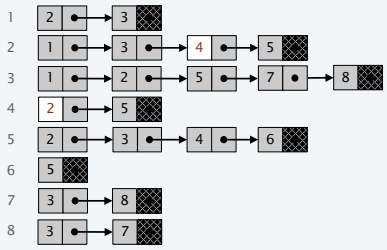
Graph representation: adjacency lists

Adjacency lists. Node indexed array of lists.

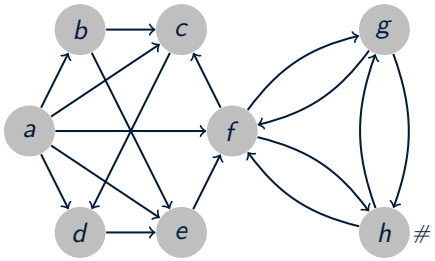
- Two representations of each edge.
- Space is $\Theta(m + n)$.
- Checking if (u, v) is an edge takes $O(degree(u))$ time.
- Identifying all edges takes $\Theta(m + n)$ time.

degree = number of neighbors of u



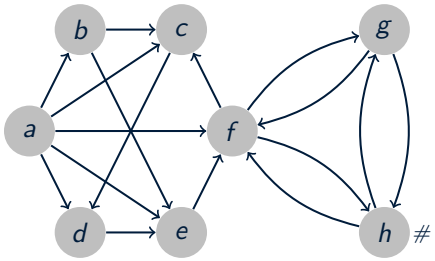


Option 1: Adjacency Sets



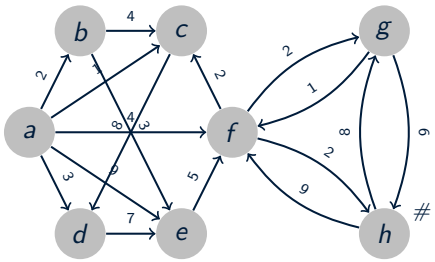

```
a, b, c, d, e, f, g, h = range(8)
N = [
    {b, c, d, e, f},     # a
    {c, e},              # b
    {d},                 # c
    {e},                 # d
    {f},                 # e
    {c, g, h},           # f
    {f, h},              # g
    {f, g}               # h
]

>>> b in N[a]
# Neighborhood membership
True
>>> len(N[f])  # Degree
3
```


Option 2: Adjacency Lists



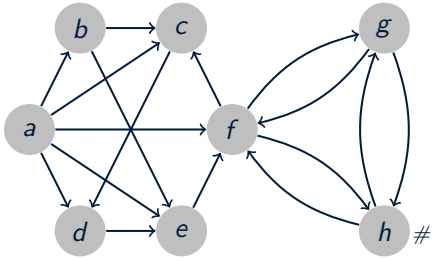

```
a, b, c, d, e, f, g, h = range(8)
N = [
    [b, c, d, e, f],     # a
    [c, e],              # b
    [d],                 # c
    [e],                 # d
    [f],                 # e
    [c, g, h],           # f
    [f, h],              # g
    [f, g]               # h
]

>>> b in N[a]
# Neighborhood membership
True
>>> len(N[f])  # Degree
3
```


Option 3: Adjacency Dictionaries w/ Edge Weights




```
a, b, c, d, e, f, g, h = range(8)
N = [
    {b:2, c:1, d:3, e:9, f:4},#a
    {c:4, e:3},               #b
    {d:8},                    #c
    {e:7},                    #d
    {f:5},                    #e
    {c:2, g:2, h:2},          #f
    {f:1, h:6},               #g
    {f:9, g:8}                #h
]

>>> b in N[a]
# Neighborhood membership
True
>>> len(N[f])  # Degree
3
>>> N[a][b]
# Edge weight for (a, b)
2
```

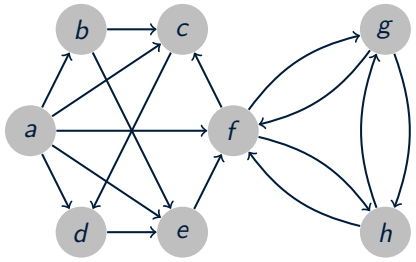

Option 4: Dictionaries w/ Adjacency Sets




```
N = {
    'a': set('bcdef'),
    'b': set('ce'),
    'c': set('d'),
    'd': set('e'),
    'e': set('f'),
    'f': set('cgh'),
    'g': set('fh'),
    'h': set('fg')
}

>>> 'b' in N['a']
# Neighborhood membership
True
>>> len(N['f'])  # Degree
3
```

Graph Data Structures

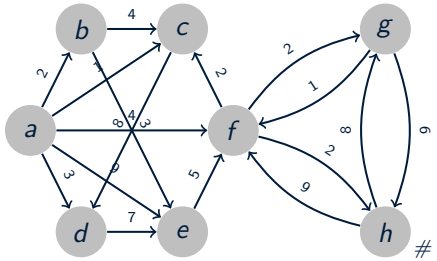
Option 5: Adjacency Matrix (using nested lists)




```
a, b, c, d, e, f, g, h = range(8)

#     a b c d e f g h

N = [[0,1,1,1,1,1,0,0], # a
     [0,0,1,0,1,0,0,0], # b
     [0,0,0,1,0,0,0,0], # c
     [0,0,0,0,1,0,0,0], # d
     [0,0,0,0,0,1,0,0], # e
     [0,0,1,0,0,0,1,1], # f
     [0,0,0,0,0,1,0,1], # g
     [0,0,0,0,0,1,1,0]] # h


>>> a, b, c, d, e, f, g, h = range(8)
>>> N[a][b]
# Neighborhood membership
1
>>> sum(N[f])  # Degree
3
```


Graph Data Structures

Option 6: Weighted Adjacency Matrix (using nested lists)




```
a, b, c, d, e, f, g, h = range(8)
_ = float('inf')

#     a  b  c  d  e  f  g  h

W = [[0,2,1,3,9,4,_,_], # a
     [_,0,4,_,3,_,_,_], # b
     [_,_,0,8,_,_,_,_], # c
     [_,_,_,0,7,_,_,_], # d
     [_,_,_,_,0,5,_,_], # e
     [_,_,2,_,_,0,2,2], # f
     [_,_,_,_,_,_,1,0,6], # g
     [_,_,_,_,_,_,9,8,0]] # h


>>> inf = float('inf')
>>> W[a][b] < inf
# Neighborhood membership
True
>>> W[c][e] < inf
# Neighborhood membership
False
>>> sum(1 for w in W[a] if w < inf) - 1
# Degree
5
```

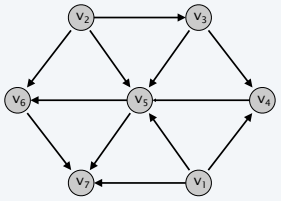

Trade-offs Between Adjacency Lists and Adjacency Matrices

- When the text refers to adjacency lists, it means adjacency linked lists
- Adjacency linked lists use less memory for sparse graphs than matrices
- Finding vertex degree is $\Theta(1)$ vs. $\Theta(n)$ in matrices
- In Python, adjacency lists are technically adjacency dynamic arrays (respectively sets and dictionaries for other representations)
- Python adjacency lists/sets/dictionaries use less memory than adjacency matrices implemented in Python
- They are also $\Theta(1)$ to find vertex degree
- Within Python, adjacency sets offer expected $\Theta(1)$ membership checking vs. $\Theta(n)$ for lists
- For graph traversal, adjacency lists execute faster than sets.

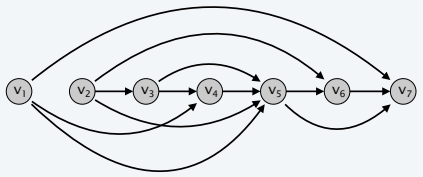
Directed acyclic graphs

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

45

Application of topological sorting

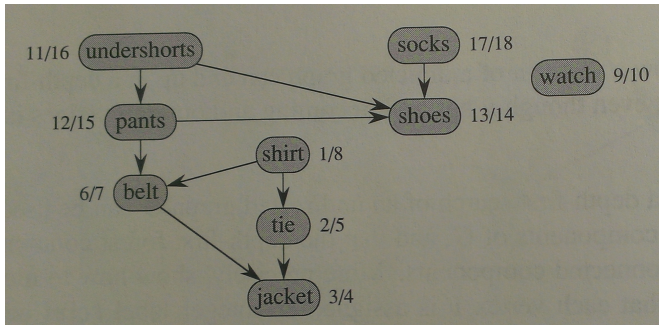


Figure: Directed acyclic graph for clothing dependencies

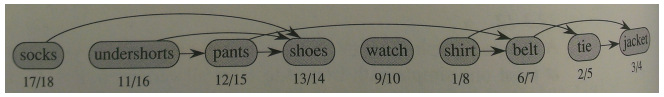


Figure: Topological sort of clothes

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.

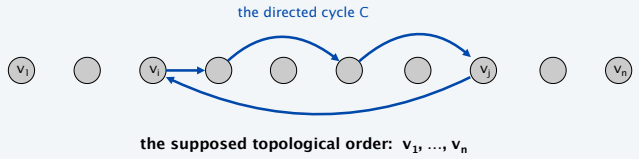
46

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. ▪



47

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?

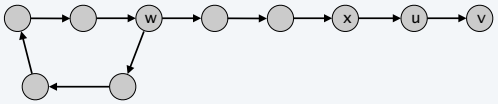
48

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. ▪



49

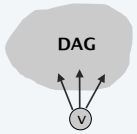
Directed acyclic graphs

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
    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 no
    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
```