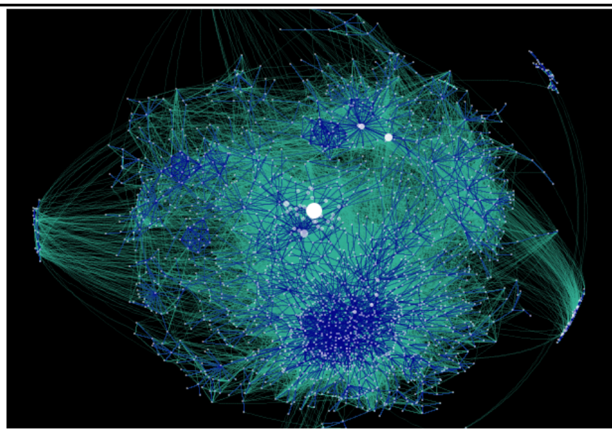
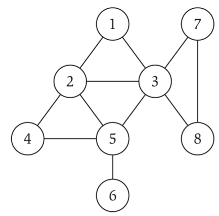
Chapter 3 - Graphs



http://datamining.typepad.com/gallery/blog-map-gallery.html

Undirected Graphs

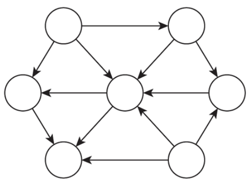
Undirected graph. G = (V, E)
- V = set of nodes.
- E = set of edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: n = |V|, m = |E|.



V = { 1, 2, 3, 4, 5, 6, 7, 8 }
E = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 }
n = 8
m = 11

Directed Graphs

Directed graph. G = (V, E)
- Edge (u, v) goes from node u to node v.



Example. Web graph - hyperlink points from one web page to another.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.

3

Graph definitions

Graph **G = (V, E)** , V: set of **nodes** or vertices, E: set of **edges** (pairs of nodes).
In an **undirected** graph, edges are unordered pairs (sets) of nodes. In a **directed** graph edges are ordered pairs (tuples) of nodes.
Path: sequence of nodes $(v_0..v_n)$ s.t. $\forall i$: (v_i, v_{i+1}) is an edge. **Path length**: number of edges in the path, or sum of weights. **Simple path:** all nodes distinct.
Cycle: path with first and last node equal. **Acyclic graph**: graph without cycles. **DAG**: directed acyclic graph.
Two nodes are **adjacent** if there is an edge between them. In a **complete graph** all nodes in the graph are adjacent.

more definitions

An undirected graph is **connected** if for all nodes v_i and v_j there is a path from v_i to v_j . An undirected graph can be partitioned in **connected components**: maximal connected sub-graphs.
A directed graph can be partitioned in **strongly connected components**: maximal sub-graphs C where for every u and v in C there is a path from u to v and there is a path from v to u.

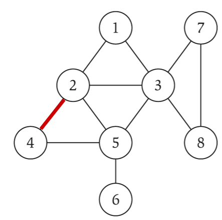
$G'(V', E')$ is a **sub-graph** of $G(V,E)$ if $V' \subseteq V$ and $E' \subseteq E$
The sub-graph of G **induced** by V' has all the edges $(u,v) \in E$ such that $u \in V'$ and $v \in V'$.

In a **weighted graph** the edges have a weight (cost, length,..) associated with them.

Graph Representation: Adjacency Matrix

Adjacency matrix. n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge, or $weight_{uv}$ in a weighted graph.

- For undirected graphs, each edge is represented **twice**.
- Space proportional to n^2.
- Checking if (u, v) is an edge takes $\Theta(1)$ time.
- Identifying all outgoing edges from a node takes $\Theta(n)$ time.



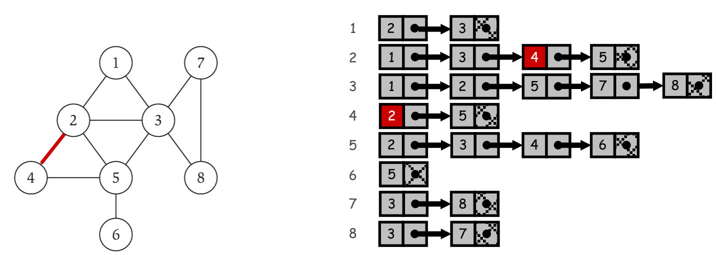
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

6

Graph Representation: Adjacency List

Adjacency list. Node indexed array of lists.

- For undirected graphs, each edge is again represented twice.
- Space proportional to m + n.
- Checking if (u, v) is an edge takes O(degree(u)) time. (degree = number of neighbors of u)
- Identifying all outgoing edges from a node takes O(degree(u)) time
- Identifying all edges takes $\Theta(m + n)$ time.
- Cool python representation: dictionary



7

Which Implementation

Which implementation best supports common graph operations:

- Is there an edge between vertex i and vertex j?
- Find all vertices adjacent to vertex j

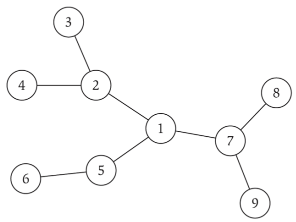
Which best uses space?

8

Trees

Def. An undirected graph is a tree if it is connected and does not contain a cycle.

How many edges does a tree have?



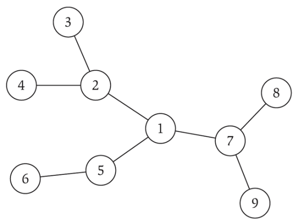
Trees

Def. An undirected graph is a tree if it is connected and does not contain a cycle.

How many edges does a tree have?

Given a set of nodes, build a tree step wise

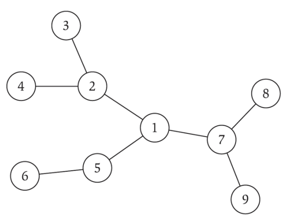
- every time you add an edge, you must add a new node to the growing tree. WHY?
- how many edges to connect n nodes?



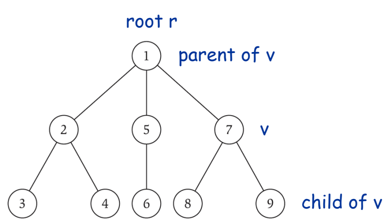
Rooted Trees

Rooted tree. Given a tree T, choose a root node r and orient each edge below r; do same for sub-trees.

Models hierarchical structure. By rooting the tree it is easy to see that it has n-1 edges.



a tree



the same tree, rooted at 1

11

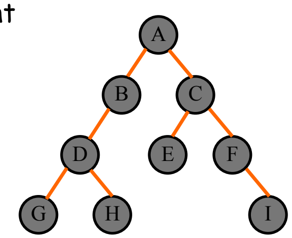
Traversing a Binary Tree

Pre order
- visit the node
- go left
- go right

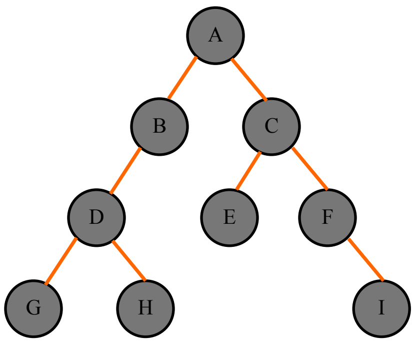
In order
- go left
- visit the node
- go right

Post order
- go left
- go right
- visit the node

Level order / breadth first
- for d = 0 to height
  - visit nodes at level d



Traversal Examples



Pre order

A B D G H C E F I

In order

G D H B A E C F I

Post order

G H D B E I F C A

Level order

A B C D E F G H I

IMPLEMENTATION of these traversals??

Tree traversal Implementation

recursive implementation of preorder

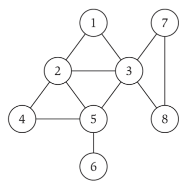
- The steps:
  - visit node
  - preorder(left child)
  - preorder(right child)
- What changes need to be made for in-order, post-order?

How would you implement level order?

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 nodes s and t, what is the length of the shortest path between s and t? Length: either in terms of number of edges, or in terms of sum of weights.



15

Graph Traversal

What makes it different from tree traversals?

Graph Traversal

What makes it different from tree traversals:
- you can visit the same node more than once
- you can get in a cycle

What to do about it?

Graph Traversal

What makes it different from tree traversals:
- you can visit the same node more than once
- you can get in a cycle

What to do about it:
- **mark** the nodes
  - -White: unvisited
  - -Grey: (still being considered) on the frontier: not all adjacent nodes have been visited yet
  - -Black: off the frontier: all adjacent nodes visited (not considered anymore)

BFS: Breadth First Search

Like **level** traversal in trees, **BFS(G,s)** explores the edges of G and locates every node v reachable from s in a level order using a queue.

BFS: Breadth First Search

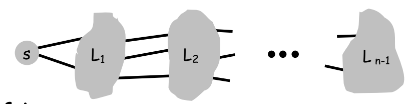
Like level traversal in trees, **BFS(G,s)** explores the edges of G and locates every node v reachable from s in a level order using a queue.
BFS also computes the **distance**: number of edges from s to all these nodes, and the **shortest path** (minimal #edges) from s to v.

BFS: Breadth First Search

Like level traversal in trees, **BFS(G,s)** explores the edges of G and locates every node v reachable from s in a level order using a queue.
BFS also computes the **distance:** number of edges from s to all these nodes, and the **shortest path** (minimal #edges) from s to v.
BFS expands a **frontier** of **discovered** but not yet visited nodes. Nodes are colored white, grey or black. They start out undiscovered or white.

Breadth First Search

BFS intuition. Explore outward from s, 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.

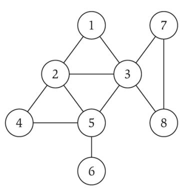
For each i, L_i consists of all nodes at distance exactly i from s. There is a path between s and t iff t appears in some layer.

22

Breadth First Tree

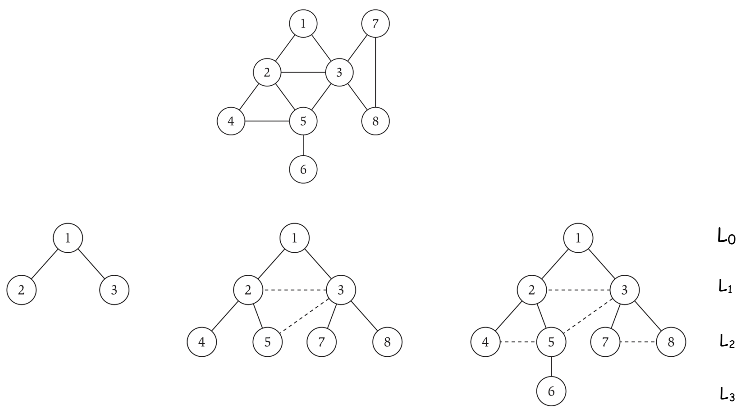
BFS produces a **Breadth First Tree** rooted at s: when a node v in L_{i+1} is discovered as a neighbor of node u in L_i we add edge (u,v) to the BF tree

Property. Let T be a BFS tree of G, and let (x, y) be an edge of G. Then the level of x and y differ by at most 1. **WHY?**



23

Breadth First Search



(a) (b) (c)

24

```
BFS(G,s)
  #d: distance,  c: color,  p: parent in BFS tree
  forall v in V-s {c[v]=white; d[v]=∞,p[v]=nil}
  c[s]=grey; d[s]=0; p[s]=nil;
  Q=empty;
  enque(Q,s);
  while (Q != empty)
     u = deque(Q);
      forall  v in adj(u)
         if (c[v]==white)
            c[v]=grey; d[v]=d[u]+1; p[v]=u;
            enque(Q,v)
     c[u]=black;
   # don't really need grey here, why?
```

Complexity BFS

Each node is painted white once, and is enqueued and dequeued at most once.

Complexity BFS

Each node is painted white once, and is enqueued and dequeued at most once.

Why?

Complexity BFS

Each node is painted white once, and is enqueued and dequeued at most once.

Enque and deque take constant time. The adjacency list of each node is scanned only once: when it is dequeued.

Complexity BFS

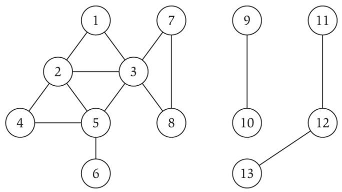
Each node is painted white once, and is enqueued and dequeued at most once.

Enque and deque take constant time. The adjacency list of each node is scanned only once, when it is dequeued.
Therefore time complexity for BFS is
$O(|V|+|E|)$ or $O(n+m)$

Connected Components

Connected graph. There is a path between any pair of nodes.

Connected component of a node s. The set of all nodes reachable from s.



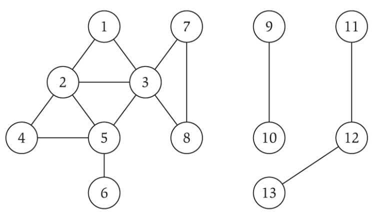
Connected component containing node 1 = { 1, 2, 3, 4, 5, 6, 7, 8 }.

30

Connected Components

Connected component of a node s. The set of all nodes reachable from s.

Given two nodes s, and t, their connected components are either identical or disjoint. **WHY?**

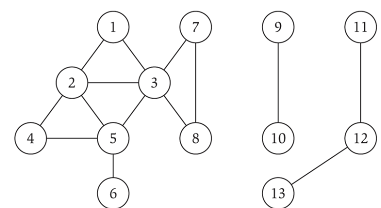


31

Connected Components

Connected component of a node s. The set of all nodes reachable from s.

Given two nodes **s**, and **t**, their connected components are either identical or disjoint.

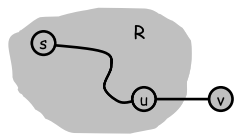


Two cases – either there is a path between s and t or there isn't.
If there is a path: take a node u in the connected component of s, and construct a path from t to u: t to s, then s to u, so $CC_s = CC_t$
If there is no path: assume that the intersection contains a node u. Use it to construct a path between s and t: s to u, then u to t – **contradiction**.

32

Connected Components

A generic algorithm for finding connected components:




```
R = {s}  # the connected component of s is initially s.
while there is an edge (u,v) where u is in R and v is not in R:
    add v to R
```

Upon termination, R is the connected component containing s.

- BFS: explore in order of distance from s.
- DFS: explores edges from the most recently discovered node; backtracks when reaching a dead-end.

DFS: Depth First Search

Explores edges from the most recently discovered node; backtracks when reaching a dead-end. The algorithm below does not use white, grey, black, but uses explored (and implicitly unexplored). Recursive code:

```
DFS(u):
  mark u as Explored and add u to R
  for each edge (u,v) :
    if v is not marked Explored :
       DFS(v)
```

BUT, how do we find cycles in a graph?

DFS and cyclic graphs

When DFS visits a node for the first time it is white. There are two ways DFS can **revisit** a node:

1. DFS has already fully explored the node. **What color does it have then? Is there a cycle then?**

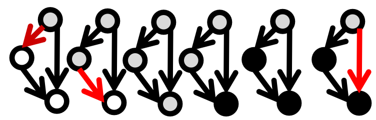
2. DFS is still exploring this node. **What color does it have in this case? Is there a cycle then?**

35

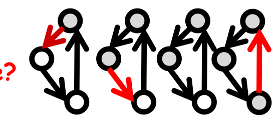
DFS and cyclic graphs

There are two ways DFS can **revisit** a node:

1. DFS has already fully explored the node. **What color does it have then? Is there a cycle then?**



2. DFS is still exploring this node. **What color does it have in this case? Is there a cycle then?**

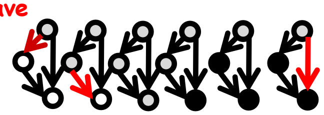


36

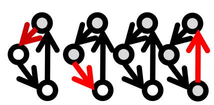
DFS and cyclic graphs

There are two ways DFS can **revisit** a node:

1. DFS has already fully explored the node. **What color does it have then? Is there a cycle then?**
No, the node is revisited from outside.



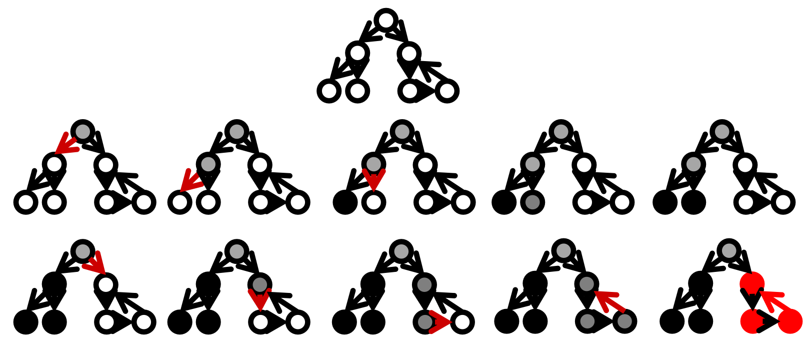
2. DFS is still exploring this node. **What color does it have in this case? Is there a cycle then?**
Yes, the node is revisited on a path containing the node itself.



So DFS with the white, grey, black coloring scheme detects a cycle when a GREY node is visited

37

Cycle detection: DFS + coloring



When a grey (frontier) node is visited, a cycle is detected.

38

Recursive / node coloring version

```
DFS(u):
    #c: color,  p: parent
    c[u]=grey
    forall  v in Adj(u):
      if c[v]==white:
        p[v]=u
        DFS(v)
    c[u]=black
```

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

Proof:

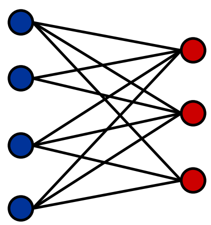
Same as in BFS ▪

Bipartite Graphs

Def. An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored red or blue such that every edge has one red end and one blue end.

Applications.

- Scheduling: machines = red, jobs = blue.



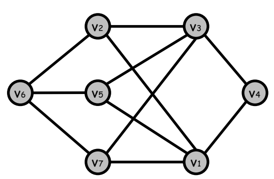
a bipartite graph

40

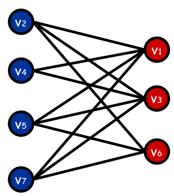
Testing Bipartite-ness

Given a graph G, is it bipartite?

- Many graph problems become tractable if the underlying graph is bipartite (independent set)
- A graph is bipartite if it is 2-colorable



a bipartite graph G

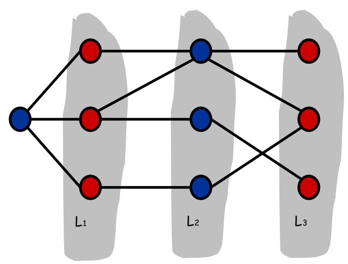


another drawing of G

41

Algorithm for testing if a graph is bipartite

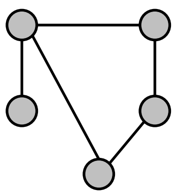
- Pick a node s and color it blue
- Its neighbors must be colored red.
- Their neighbors must be colored blue.
- Proceed until the graph is colored.
- Check that there is no edge whose ends are the same color.

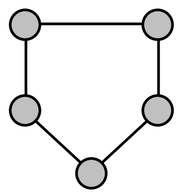


42

An Obstacle to Bipartite-ness

Which of these graphs is 2-colorable?

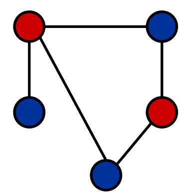




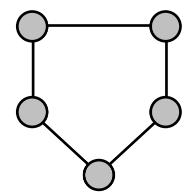
An Obstacle to Bipartite-ness

Lemma. If a graph *G* is bipartite, it cannot contain an odd cycle.

Proof. Not possible to 2-color the odd cycle, let alone *G*.



bipartite
(2-colorable)



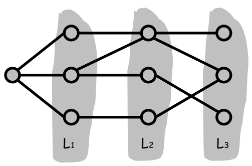
not bipartite
(not 2-colorable)

Bipartite Graphs

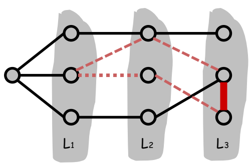
Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer. G is bipartite.

(ii) An edge of G joins two nodes of the same layer. G contains an odd-length cycle (and hence is not bipartite).



Case (i)



Case (ii)

45

Bipartite Graphs

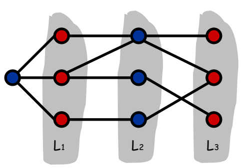
Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer. G is bipartite.

(ii) An edge of G joins two nodes of the same layer. G contains an odd-length cycle (and hence is not bipartite).

Proof. (i)

- Suppose no edge joins two nodes in the same layer.
- I.e. all edges join nodes on adjacent layers.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.



Case (i)

46

Bipartite Graphs

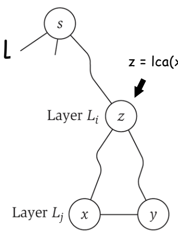
Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer. G is bipartite.
(ii) An edge of G joins two nodes of the same layer. G contains an odd-length cycle (and hence is not bipartite).

Proof. (ii)

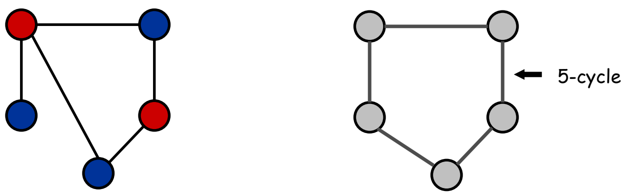
- Suppose (x, y) is an edge with x, y in same level L_j.
- Let z = lca(x, y) = lowest common ancestor.
- Let L_i be the level containing z.
- Consider the cycle containing the edge (x,y), then path from y to z, then path from z to x.
- Its length is 1 + (j-i) + (j-i), which is odd. ▪

(x, y) — path from y to z — path from z to x



Obstruction to Bipartiteness

Corollary. A graph G is bipartite iff it contains no odd length cycle.

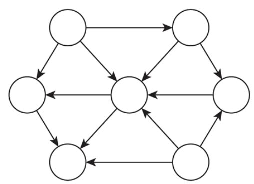


bipartite
(2-colorable)

not bipartite
(not 2-colorable)

Directed Graphs

Directed graph. $G = (V, E)$
- Edge (u, v) goes from node u to node v.



Example. Web graph - hyperlink points from one web page to another.
- Search engines exploit hyperlink structure to rank web pages by importance.

Graph Search

Directed reachability. Given a node s, find all nodes reachable from s.

Web crawler. Start from web page s. Find all web pages linked from s, either directly or indirectly.

BFS and DFS extend naturally to directed graphs.

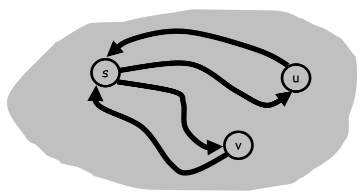
Given a path from s to t, not guaranteed there is a path from t to s.

Strong Connectivity

Def. Nodes u and v are mutually reachable if there is a path from u to v and also a path from v to u.
Def. A graph is strongly connected if every pair of nodes is mutually reachable.

Lemma. Let s be any node. G is strongly connected iff every node is reachable from s, and reversely, s is reachable from every node.

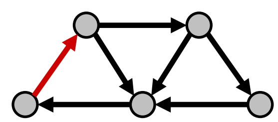
Proof. ⇒ Follows from definition.
Proof. ⇐ Path from u to v: concatenate u-s path with s-v path.
Path from v to u: concatenate v-s path with s-u path. ▪



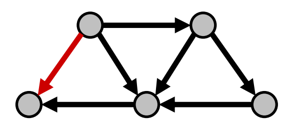
51

Strong Connectivity: Algorithm

Theorem. Strong connectivity of a graph can be determined in O(m + n) time.
Proof.
- Pick any node s.
- 1. Run BFS from s in G.
- 2. Run BFS from s in G^{rev}. (reverse orientation of every edge in G)
- Return true iff all nodes reached in both BFS executions.
- 1: s can reach all nodes, 2: s can be reached from all nodes
- Correctness follows immediately from previous lemma. ▪



strongly connected



not strongly connected

52

DAGs and Topological Ordering

Examples: Graphs Describing Precedence

- prerequisites for a set of courses
- dependencies between programs
- dependencies between jobs
- order of putting your clothes on

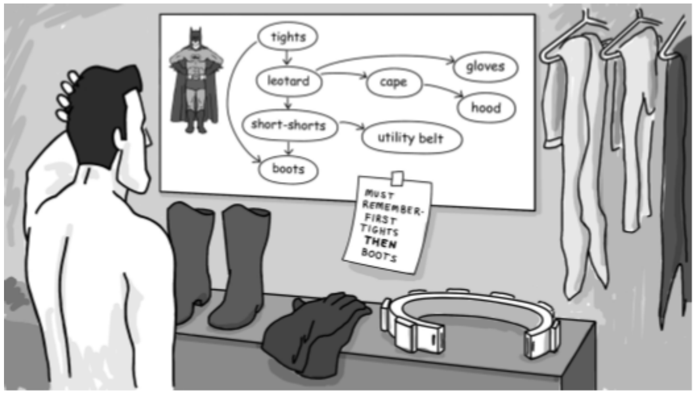
Precedence constraints. Edge (v_i, v_j) means task v_i must occur before v_j.

Topological ordering: a total ordering of the nodes that respects the precedence relation

- Example: An ordering of CS courses

Graphs describing precedence must not contain cycles.
Why?

Graphs Describing Precedence

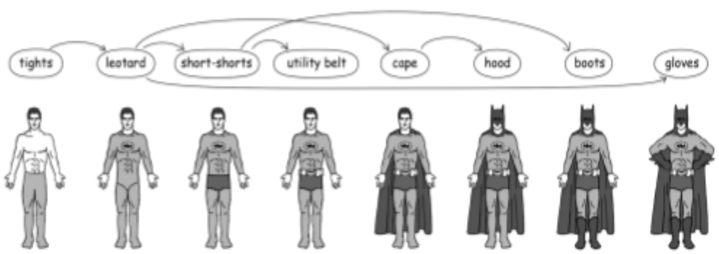


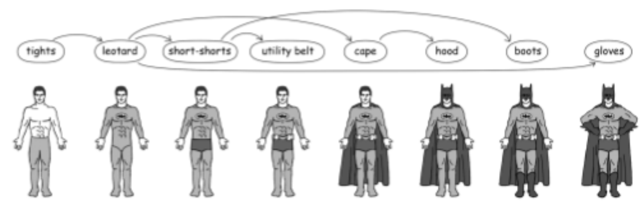
Batman images are from the book “Introduction to bioinformatics algorithms”

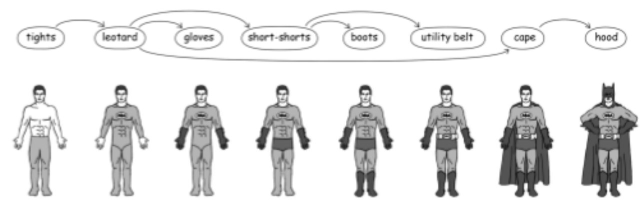
Topological Order of DAGs

DAG: Directed Acyclic Graph

Topological order: listing of nodes such that if (a,b) is an edge, a appears before b in the list
Is a topological sort unique?







Directed Acyclic Graphs

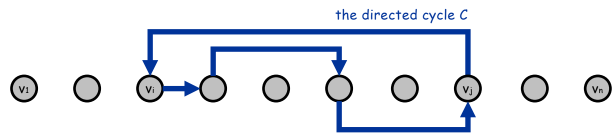
Lemma. If G has a topological order, then G is a DAG.

Directed Acyclic Graphs

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

Proof. (by contradiction)

- Suppose that G has a topological order $v_1, \ldots, v_n$ and that G also has a directed cycle C.
- 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 $v_1, \ldots, v_n$ is a topological order, we must have $j < i$
- **contradiction.**



the supposed topological order: $v_1, \ldots, v_n$

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

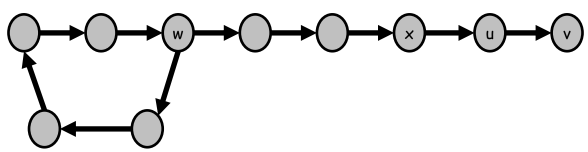
Lemma. If G is a DAG, then G has a node with no incoming edges.

Directed Acyclic Graphs

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

Proof. (by contradiction)

- Suppose that G is a DAG and every node has at least one incoming edge. Then it has a cycle and thus is not a DAG:
  - Pick any node v, and begin following edges backward from v.
  - Repeat. After n + 1 steps we have visited a node, say w, twice. The sequence of nodes encountered between successive visits is a cycle.
- **Contradiction**



61

Directed Acyclic Graphs

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

Proof. (by induction on n)

- Base: true if n = 1.
- Step: Given a DAG with n > 1 nodes, find a node v with no incoming edges. G - { v } is a DAG, since deleting v cannot create cycles. By induction hypothesis, G - { v } has a topological ordering. ▪

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

62

Topological Sort: Algorithm

Algorithm:

keep track of # incoming edges per node
while (nodes left) :
    extract one with 0 incoming
    subtract one from all its adjacent nodes

Time complexity?
Better way?

63

Topological Sort: Algorithm Running Time

Theorem. Algorithm can run in $O(m + n)$ time.
Proof.

- Maintain the following information:
  - `count[w]` = remaining number of incoming edges
  - S = set of nodes with no incoming edges
- Initialization: $O(m + n)$ via single scan through graph.
- Update: pick a node v in S
  - remove v from S
  - for each edge (v, w) : decrement `count[w]` and add w to S if `count[w]` hits 0
  - this is $O(1)$ per edge ▪

64