EE360C: Algorithms

Graphs

Pedro Santacruz

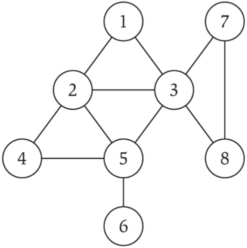
Department of Electrical and Computer Engineering
University of Texas at Austin

Fall 2014

Undirected Graphs

Undirected graph: $G = (V, E)$

- V: nodes
- E: edges between pairs of nodes
- captures pairwise relationships between objects
- graph size parameters: $n = |V|$, $m = |E|$

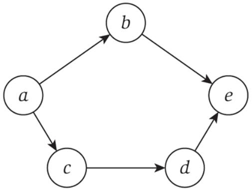


- $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- $E = \{\{1,2\}, \{1,3\}, \{2,3\}, \{2,4\}, \{2,5\}, \{3,5\}, \{3,7\}, \{3,8\}, \{4,5\}, \{5,6\}\}$
- $n = 8$
- $m = 11$

Directed Graphs

Directed graph: $G = (V, E)$

- V: nodes
- E: edges between pairs of nodes
- captures one-way relationships between objects
- graph size parameters: $n = |V|$, $m = |E|$



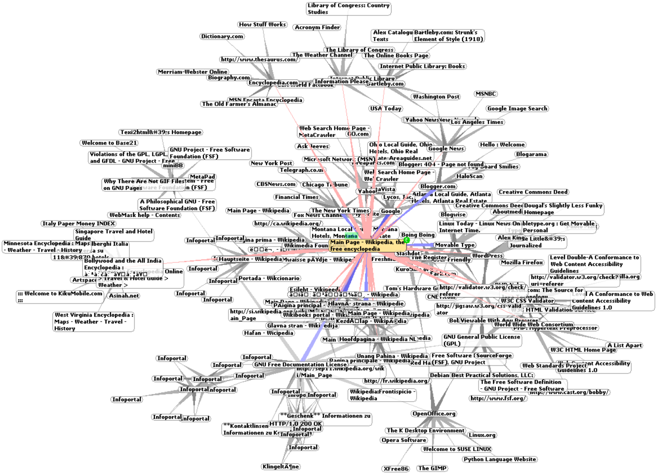
- $V = \{a, b, c, d, e\}$
- $E = \{\{a, b\}, \{a, c\}, \{b, e\}, \{c, d\}, \{d, e\}\}$
- $n = 5$
- $m = 5$

Example Graph Applications

Graph	Nodes	Edges	Directed
transportation	intersections	highways	no
communication	computers	fiber optic cables	no
World Wide Web	web pages	hyperlinks	yes
social	people	relationships	maybe
food web	species	predator/prey	yes
software systems	functions	function calls	yes
scheduling	tasks	precendences	yes

World Wide Web

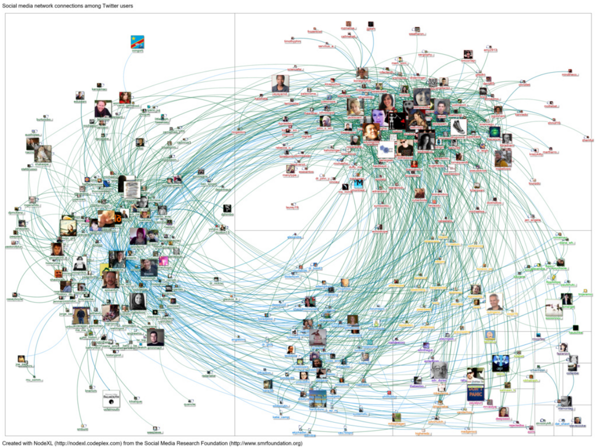
- nodes: webpages
- edges: hyperlinks



Chris 73/Wikimedia Commons
http://en.wikipedia.org/wiki/File:WorldWideWebAroundWikipedia.png

Social Network

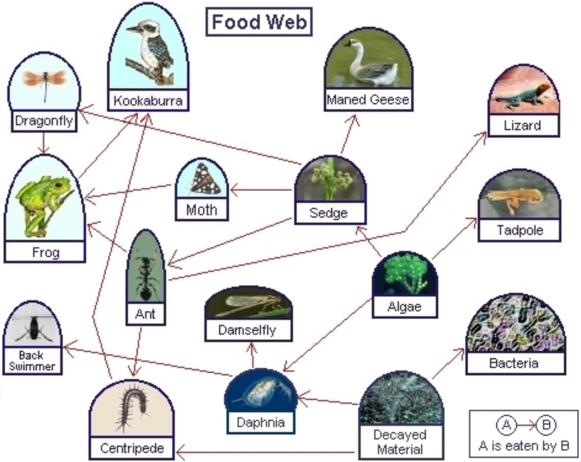
- nodes: people
- edges: relationships between people



www.connectedaction.net, Author: Marc Smith

ECOLOGICAL FOOD WEB

- nodes: animals and plants
- edges: eating relationship



http://cbc.amnh.org/crisis/foodweb.html

Graph Representation

There are basically two common ways of representing a graph $G = (V, E)$:

- a collection of *adjacency lists*
- *adjacency matrix*

Generally, we prefer the adjacency list representation because it uses considerably less memory for the more common *sparse graphs* (i.e., when $m \ll n^2$). We prefer the adjacency matrix representation when either:

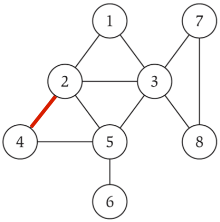
- the graph is *dense*, i.e., m is close to n^2
- we need to quickly check if a particular edge exists

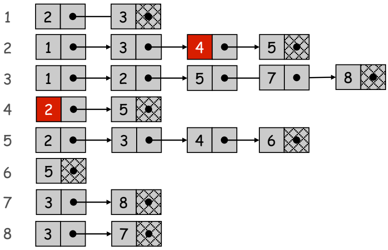
Adjacency List Representation

Given a graph $G = (V, E)$, we represent it by an array Adj of n lists, one for each vertex in V.

- for each $u \in V$, $\mathrm{Adj}[u]$ is a list of vertices v such that there is an edge from u to v
- the order of the list is arbitrary
- for a directed graph, $\sum_{u \in V} |\mathrm{Adj}[u]| = m$
- for an undirected graph, $\sum_{u \in V} |\mathrm{Adj}[u]| = 2m$
- in either case, the total memory required to represent the graph is $\Theta(n + m)$

Adjacency Lists Visualized





Adjacency Lists and Weights

We often want to store weighted graphs; the adjacency list representation lends itself well to this:

- in a weighted graph, each edge has an associated weight, i.e., $w : E \to \mathbf{R}$
- we can easily store $w(u, v)$ in an adjacency list representation

Adjacency Matrix Representation

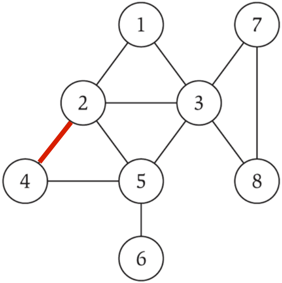
One problem with adjacency lists is that to determine if there's an edge from u to v, we have to search u's entire list. This can be expensive if there are a lot of edges.

An adjacency matrix represents a graph $G = (V, E)$ using a $n \times n$ matrix $A = (a_{ij})$, such that $a_{ij} = 1$ if $(i, j) \in E$ and 0 otherwise.

- this requires $\Theta(n^2)$ memory, regardless of the number of edges.
- given that A^T is the transpose of A, i.e., $a^T_{ij} = a_{ji}$, for an undirected graph, $a_{ij} = a_{ji}$, and we can use half of the storage space

For a weighted graph, instead of storing 1 in a_{ij}, we store the weight of the edge (i, j).

Adjacency Matrix Visualized



	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	1	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

Question

Given an adjacency matrix representation of a graph, what is the time complexity for checking if the edge (u, v) exists in the graph?

Given an adjacency list representation of a graph, what is the time complexity for checking if the edge (u, v) exists in the graph?

Paths and Connectivity

Definition: Path

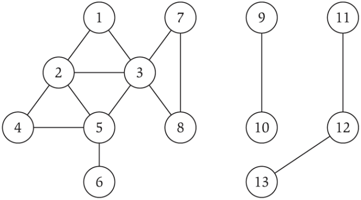
A path in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair $v_i, v_i + 1$ is joined by an edge in E.

Definition: Simple Path

A path P is simple if all nodes in P are distinct.

Definition: A Connected Undirected Graph

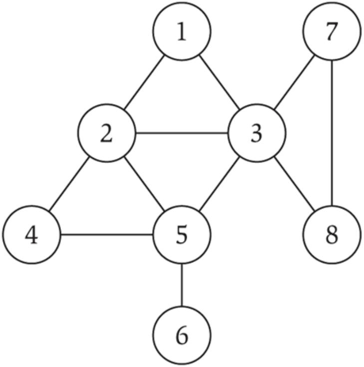
An undirected graph is connected if for every pair of nodes u and v, there is a path between u and v.



CYCLES

Definition: Cycle

A cycle is a path $v_1, v_2, \ldots v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k - 1$ nodes are all distinct.



cycle $= 1 - 2 - 4 - 5 - 3 - 1$

Trees

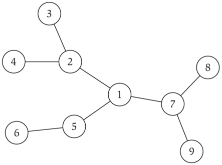
Definition: Tree

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

Theorem

Let G be an undirected graph on n nodes. Any two of the following statements imply the third:

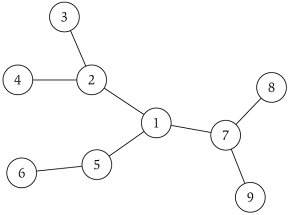
- G is connected
- G does not contain a cycle
- G has $n - 1$ edges



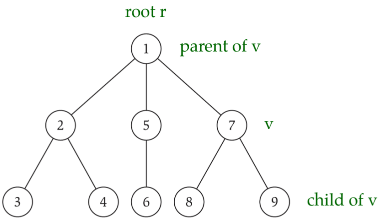
ROOTED TREES

Definition: Rooted Tree

Given a tree T, choose a root node r and orient each edge away from r. This enables one to model hierarchical structure.

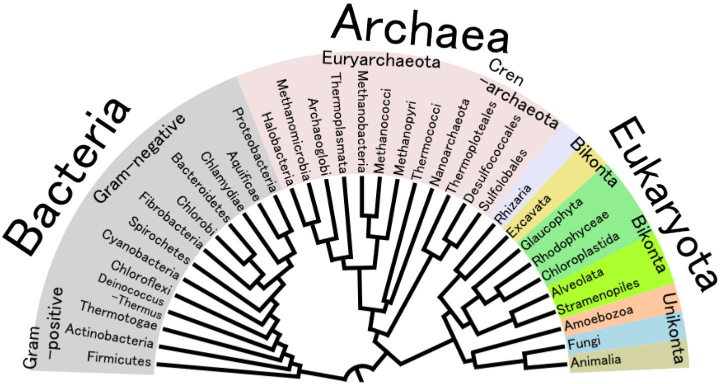


a tree



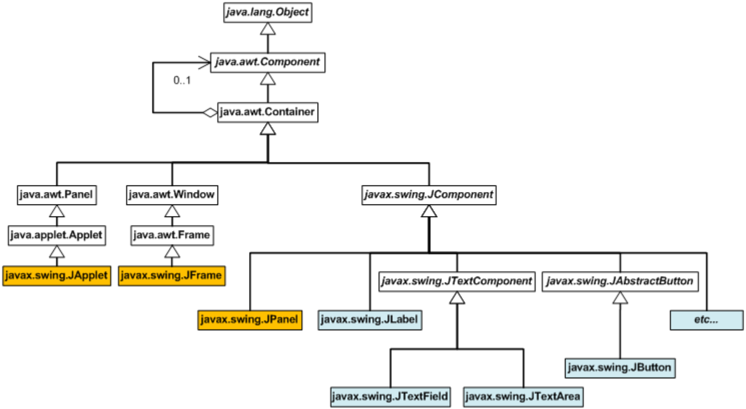
the same tree, rooted at 1

An Example Tree: Phylogeny Tree



http://en.wikipedia.org/wiki/File:Phylogenetic_Tree_of_Life.png

Another Example Tree: Object Oriented Class Architecture



http://www.clear.rice.edu/comp310/JavaResources/GUI/

Connectivity

$s - t$ Connectivity Problem

Given two nodes 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?

Applications

- Social network connections (e.g., Kevin Bacon number)
- Maze traversal
- Fewest number of hops in a communication network

Breadth First Search

The Problem

Given a graph $G = (V, E)$ and a specific source vertex s, what vertices can be reached from s?

Not only is this problem pretty pervasive (e.g., in task scheduling), it is also a basis for other more advanced graph algorithms.

The basic idea is to systematically explore the edges of G to "discover" each node reachable from s.

- this works for both directed and undirected graphs
- the name of BFS comes from the fact that it expands the search for new nodes uniformly across the "frontier" of discovered nodes

Breadth-First Search Conceptually

In a BFS, you can think of all nodes as being colored either white, gray, or black. Initially, all nodes are white.

- a node is "discovered" the first time the BFS encounters it; at this point BFS colors the node gray
- the complete set of gray nodes is the "frontier"
- to proceed, BFS looks at each of the gray nodes, examines each of its outgoing edges, to see if they're connected to any white (undiscovered) nodes
    - if so, color that node gray and insert this node at the **end** of the queue of the frontier vertices
    - when we've examined all of a node's outgoing edges, remove it from the frontier queue and color it black

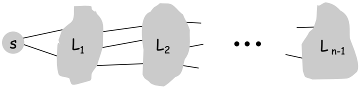
Breadth-First Search in Layers

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 if and only iff t appears in some layer.

Breadth First Search and Adjacency

Theorem

Let T be a breadth first search tree, let x and y be nodes in T belonging to layers L_i and L_j respectively, and let (x, y) be an edge of G. Then i and j differ by at most 1.

Proof

??

Breadth First Search Analysis

Assume that we use a queue to keep track of available "discovered" but unexplored nodes and adjacency lists to store the graph.

All nodes are initially undiscovered.

- each node is discovered at most once; queue operations are $O(1)$ at most; at most $O(n)$ time is spent interacting with the queue
- each adjacency list is scanned at most once (when the node is explored); so the total time spent looking at adjacency lists is $O(2m) = O(m)$

So the total running time of breadth first search is $O(n + m)$, or linear in size to the adjacency list representation.

BFS and Shortest Paths

The level of a node in a breadth first search is the *distance* computed by the breadth first search algorithm from s to u.

We define the **shortest-path distance**, $d(s, v)$ from s to v as the minimum number of edges in any path from s to v

- if there is no path from s to v, then $d(s, v) = \infty$

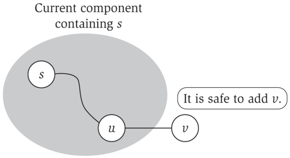
It is a non-trivial fact that the levels computed in breadth first search are the shortest distances from s to any node u. We'll revisit this problem in Chapter 4.

CONNECTED COMPONENTS

A Related Problem: Finding Connected Components

Find all nodes that are reachable from s.

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```



Theorem

Upon termination, R is the connected component containing s.

Proof

??

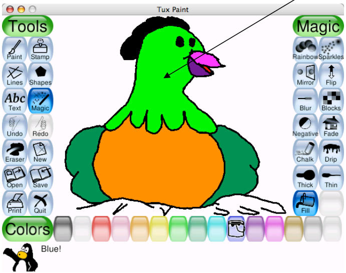
Connected Components: Practically

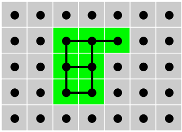
Flood Fill

Given a lime green pixel in an impage, change the color of the 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





Depth-First Search

An alternative to exploring across the entire frontier at the same time is to explore a single path as far as it can go, then explore a different one.

- depth-first search explores "deeper" into the graph whenever possible
- edges are explored out of the most recently discovered vertex (v) until there are no more
- then the search backtracks, exploring other paths out of v's parent

Depth-First Search Continued


```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u,v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

Theorem

Let T be a depth-first search tree, let x and y be nodes in T, and let (x, y) be an edge of G that is not an edge of T. Then one of x or y is an ancestor of the other in T.

Proof

??

Hint

Use the fact that, for a given recursive call $DFS(u)$, all nodes that are marked "Explored" between the invocation and the end of this recursive call are descendants of u in T.

Comparing BFS and DFS

Similarities

- Both build the strongly connected component of G that contains s.
- Both have similar efficiency

Differences

- They explore the vertices of G in very different orders.
- They result in trees rooted at s that have very different structure (bushy vs. tall)

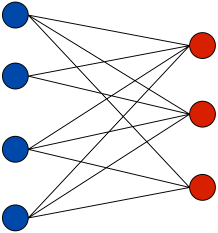
Bipartite Graphs

Definition

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

Applications

- Stable marriage: men = red, women = blue
- Scheduling: machines = red, jobs = blue

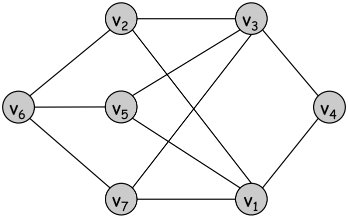


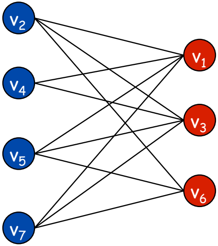
Testing Bipartiteness

Testing Bipartiteness

Given a graph G, is it bipartite?

- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)
- Before attempting to design an algorithm, we need to understand the structure of bipartite graphs





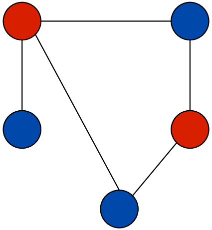
Proofs About Bipartiteness

Lemma

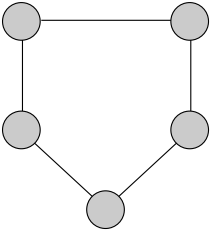
If a graph G is bipartite, it cannot contain an odd length cycle.

Proof Sketch

It is not possible to "2-color" the odd cycle (let alone the entire graph G)



bipartite
(2-colorable)



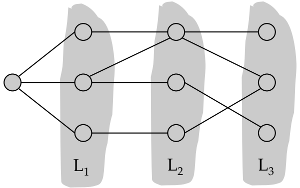
not bipartite
(not 2-colorable)

Bipartite Graphs

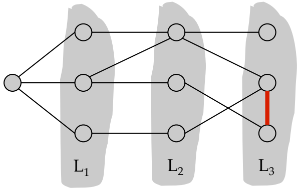
Lemma

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.

1. No edge of G joins two nodes of the same layer, and G is bipartite.
2. An edge of G joins two nodes in the same layer, and G contains an odd length cycle (and hence is not bipartite).



Case 1



Case 2

Bipartite Graphs

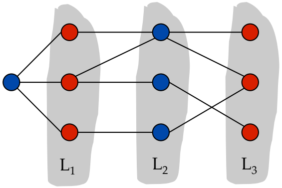
Lemma

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.

1. No edge of G joins two nodes of the same layer, and G is bipartite.
2. An edge of G joins two nodes in the same layer, and G contains an odd length cycle (and hence is not bipartite).

Proof (Case 1)

Suppose no edge joins two nodes in the same layer. By the previous lemma, this implies that all edges join nodes on adjacent levels. Then the bipartition is such that nodes on odd levels are red; nodes on even levels are blue.



Case 1

Bipartite Graphs

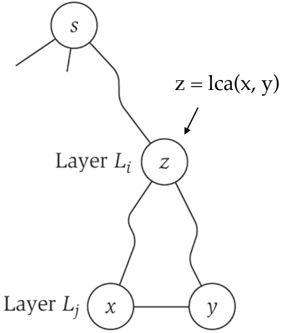
Lemma

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.

1. No edge of G joins two nodes of the same layer, and G is bipartite.
2. An edge of G joins two nodes in the same layer, and G contains an odd length cycle (and hence is not bipartite).

Proof (Case 2)

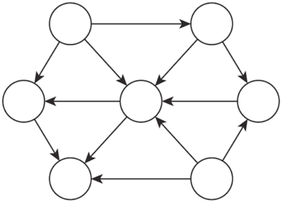
Suppose (x, y) is an edge with x and y in the same level L_j. Let z be the lowest common ancestor of x and y. Let L_i be the level containing z. Consider the cycle that takes the edge from x to y, then the path from y to z, then the path from z to x. It's length is $1 + (j - i) + (j - i)$, which is odd.



Directed Graphs

Directed Graph

In a directed graph, $G = (V, E)$, an edge (u, v) goes from node u to node v.



Example

In a web-graph, hyperlinks point *from* one web page *to* another.

- Directedness of the graph is crucial.
- Modern web search engines exploit the hyperlink structure to rank web pages by importance.

Graph Search

Directed Reachability

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

Directed $s - t$ Shortest Path Problem

Given two nodes s and t, what is the length of the shortest path between s and t?

Graph Search

Breadth first search (and depth first search) extend naturally to directed graphs.

Web Crawler

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

Strong Connectivity

Definition

Node u and v are mutually reachable if there is a path from u to v and also a path from v to u.

Definition

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 s is reachable from every node.

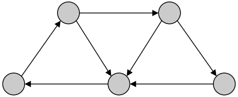
Determining Strong Connectivity

Theorem

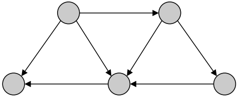
We can determine if G is strongly connected in $O(m + n)$ time.

Algorithm

- Pick any node s.
- Run BFS from s in G.
- Run BFS from s in G_{rev} (the reverse orientation of every edge in G)
- Return true iff all nodes reached in both BFS executions
- Correctness follows from the previous lemma



strongly connected



not strongly connected

Strong Components

Definition

The strong component containing a node s in a directed graph is the set of all v such that s and v are mutually reachable.

The previous algorithm is really computing the strong component containing s.

Theorem

For any two nodes s and t in a directed graph, their strong components are either identical or disjoint.

Directed Acyclic Graphs

Definition

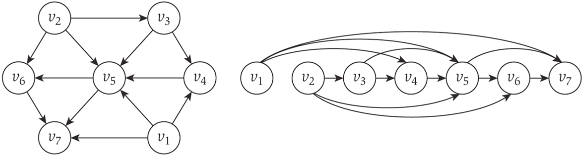
A DAG is a directed graph that contains no directed cycles.

Example

Precedence constraints: edge (v_i, v_j) means v_i must precede v_j.

Definition

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 $i > j$.



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

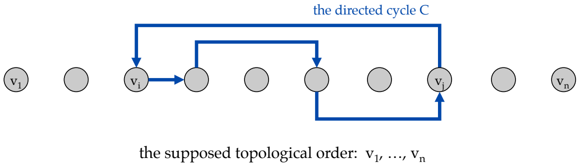
DAGs and Topological Sort

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.
- Let v_i be the lowest-indexed node in the cycle and let v_j be the node just before v_i. Thus (v_j, v_i) is an edge in E.
- 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 $j < i$, a contradiction.



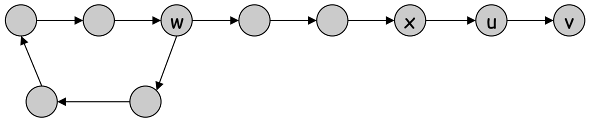
DAGs and Topological Sort (cont.)

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.
- Pick any node v and begin following edges backward from v. Since v has at least one incoming edge (u, v), we can walk backward to u.
- Then since u has at least one incoming 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.



Computing a Topological Ordering

Lemma

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

Proof (by induction)

- Base case: true if $n = 1$.
- Given a DAG on $n > 1$ nodes, find a node v with no incoming 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 the topological ordering, then append the nodes of $G - \{v\}$ in topological order. This is valid since v has no incoming 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

Topological Sort Analysis

Theorem

The algorithm finds a topological order in $O(m+n)$ time.

Proof

- Maintain the following information:
  - `count[w]`: the remaining number of incoming edges
  - S: the set of remaining nodes with no incoming edges
- Initialization: $O(m+n)$ via a single scan through the graph
- Update: to delete v:
  - remove v from S
  - decrement `count[w]` for all edges v to w, and add w to S if `count[w]` hits 0
  - In aggregate, this is constant time per edge

QUESTIONS

?