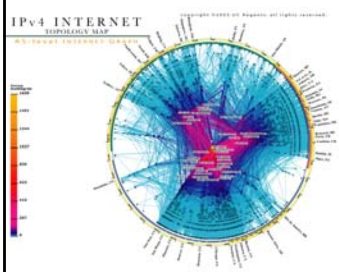
Graphs



Lecture 21
CS211 – Fall 2006

Prelim 2 Reminder

- Prelim 2
  - Tuesday, Nov 14, 7:30-9pm
  - One week from today!
  - Topics: all material through Nov 2
  - Does *not* include
    - Graphs
    - GUIs in Java
- Exam conflicts
  - Email Kelly Patwell (ASAP)
- Note: this week's Section meetings are last chance to ask questions about exam
  - Next week = regular section meeting
- Prelim 2 Review Session
  - Time & room are not yet determined
  - See *Exams* on course website for up-to-date information
  - Individual appointments are available if you cannot attend the review session (email *one* TA to arrange appointment)
- Old exams are available for review on the course website

Prelim 2 Topics

- Asymptotic complexity
- Searching and sorting
- Basic ADTs
  - stacks
  - queues
  - sets
  - dictionaries
  - priority queues
- Basic data structures used to implement these ADTs
  - arrays
  - linked lists
  - hash tables
  - BSTs
  - balanced BSTs
  - heaps
- Know and understand the sorting algorithms
  - From lecture
  - From text (not Shell Sort)
- Know the algorithms associated with the various data structures
  - Know BST algorithms, but don't need to memorize *balanced* BST algorithms
- Know the runtime tradeoffs among data structures
- Don't worry about details of JCF
  - But should have basic understanding of what's available

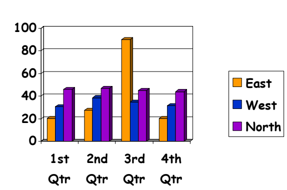
Data Structure Runtime Summary

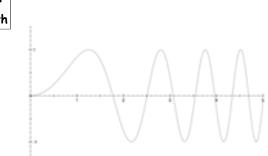
- Stack [ops = put & get]
  - O(1) worst-case time
    - Array (but can overflow)
    - Linked list
  - O(1) expected time
    - Array with doubling
- Queue [ops = put & get]
  - O(1) worst-case time
    - Array (but can overflow)
    - Linked list (need to keep track of both head & last)
  - O(1) expected time
    - Array with doubling
- Priority Queue [ops = insert & getMin]
  - O(1) worst-case time
    - Bounded height PQ (only works if few priorities)
  - O(log n) worst-case time
    - Heap (but can overflow)
    - Balanced BST
  - O(log n) expected time
    - Heap (with doubling)
  - O(n) worst-case time
    - Unsorted linked list
    - Sorted linked list (O(1) for getMin)
    - Unsorted array (but can overflow)
    - Sorted array (O(1) for getMin, but can overflow)

Data Structure Runtime Summary (Cont'd)

- Set [ops = insert & remove & contains]
  - O(1) worst-case time
    - Bit-vector (can also do union and intersect in O(1) time)
  - O(1) expected time
    - Hash table (with doubling & chaining)
  - O(log n) worst-case time
    - Balanced BST
  - O(n) worst-case time
    - Linked list
    - Unsorted array
    - Sorted array (O(log n) for contains)
- Dictionary [ops = insert(k,v) & get(k) & remove(k)]
  - O(1) expected time
    - Hash table (with doubling & chaining)
  - O(log n) worst-case time
    - Balanced BST
  - O(log n) expected time
    - Unbalanced BST (if data is sufficiently random)
  - O(n) worst-case time
    - Linked list
    - Unsorted array
    - Sorted array (O(log n) for contains)

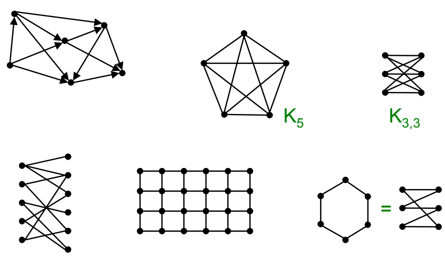
These are not Graphs





...not the kind we mean, anyway

These are Graphs



Applications of Graphs

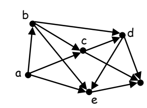
- Communication networks
- Routing and shortest path problems
- Commodity distribution (flow)
- Traffic control
- Resource allocation
- Geometric modeling
- ...

Graph Definitions

- A directed graph (or digraph) is a pair (V, E) where
  - V is a set
  - E is a set of ordered pairs (u,v) where $u,v \in V$
    - Usually require $u \neq v$ (i.e., no self-loops)
- An element of V is called a vertex (pl. vertices) or node
- An element of E is called an edge or arc
- |V| = size of V, often denoted n
- |E| = size of E, often denoted m

Example Directed Graph

Example:



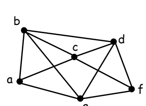
V = {a,b,c,d,e,f}
E = {(a,b), (a,c), (a,e), (b,c), (b,d), (b,e), (c,d), (c,f), (d,e), (d,f), (e,f)}

|V| = 6, |E| = 11

Example *Undirected* Graph

An *undirected graph* is just like a directed graph, except the edges are *unordered pairs* (*sets*) {u,v}

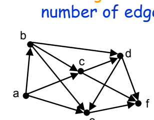
Example:

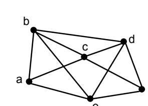


V = {a,b,c,d,e,f}
E = {{a,b}, {a,c}, {a,e}, {b,c}, {b,d}, {b,e}, {c,d}, {c,f}, {d,e}, {d,f}, {e,f}}

Some Graph Terminology

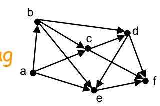
- Vertices u and v are called the source and sink of the directed edge (u,v), respectively
- Vertices u and v are called the endpoints of (u,v)
- Two vertices are adjacent if they are connected by an edge
- The outdegree of a vertex u in a directed graph is the number of edges for which u is the source
- The indegree of a vertex v in a directed graph is the number of edges for which v is the sink
- The degree of a vertex u in an undirected graph is the number of edges of which u is an endpoint



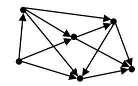


More Graph Terminology

- A path is a sequence $v_0, v_1, v_2, \ldots, v_p$ of vertices such that $(v_i, v_{i+1}) \in E$, $0 \le i \le p - 1$
- The length of a path is its number of edges
  - In this example, the length is 5
- A path is simple if it does not repeat any vertices
- A cycle is a path $v_0, v_1, v_2, \ldots, v_p$ such that $v_0 = v_p$
- A cycle is simple if it does not repeat any vertices except the first and last
- A graph is acyclic if it has no cycles
- A directed acyclic graph is called a dag

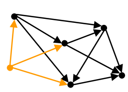


Is this a dag?



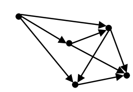
- Intuition:
  - If it's a dag, there should be a "first" vertex (i.e., a vertex with indegree zero)
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Is this a dag?



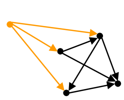
- Intuition:
  - If it's a dag, there should be a "first" vertex (i.e., a vertex with indegree zero)
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Is this a dag?



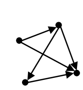
- Intuition:
  - If it's a dag, there should be a "first" vertex (i.e., a vertex with indegree zero)
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Is this a dag?



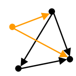
- Intuition:
  - If it's a dag, there should be a "first" vertex (i.e., a vertex with indegree zero)
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Is this a dag?



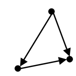
- Intuition:
  - If it's a dag, there should be a "first" vertex (i.e., a vertex with indegree zero)
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Is this a dag?



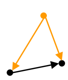
- Intuition:
  - If it's a dag, there should be a "first" vertex (i.e., a vertex with indegree zero)
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Is this a dag?



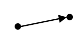
- Intuition:
  - If it's a dag, there should be a "first" vertex (i.e., a vertex with indegree zero)
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Is this a dag?



- Intuition:
  - If it's a dag, there should be a "first" vertex (i.e., a vertex with indegree zero)
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

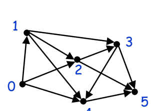
Is this a dag?



- Intuition:
  - If it's a dag, there should be a "first" vertex (i.e., a vertex with indegree zero)
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Topological Sort

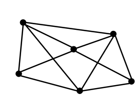
- We just computed a topological sort of the dag
  - This is a numbering of the vertices such that all edges go from lower- to higher-numbered vertices



Useful in job scheduling with precedence constraints

Graph Coloring

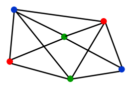
- A coloring of an undirected graph is an assignment of a color to each node such that no two adjacent vertices get the same color



- How many colors are needed to color this graph?

Graph Coloring

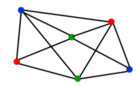
- A coloring of an undirected graph is an assignment of a color to each node such that no two adjacent vertices get the same color



- How many colors are needed to color this graph?
  - 3

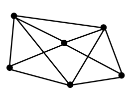
An Application of Coloring

- Vertices are jobs
- Edge (u,v) is present if jobs u and v each require access to the same shared resource, and thus cannot execute simultaneously
- Colors are time slots to schedule the jobs
- Minimum number of colors needed to color the graph = minimum number of time slots required



Planarity

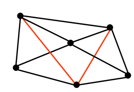
- A graph is planar if it can be embedded in the plane with no edges crossing



- Is this graph planar?

Planarity

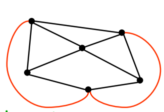
- A graph is planar if it can be embedded in the plane with no edges crossing



- Is this graph planar?
  - Yes

Planarity

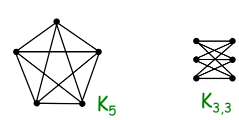
- A graph is planar if it can be embedded in the plane with no edges crossing



- Is this graph planar?
  - Yes

Detecting Planarity

Kuratowski's Theorem



A graph is planar if and only if it does not contain a copy of K_5 or $K_{3,3}$ (possibly with other nodes along the edges shown)

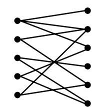
The Four-Color Theorem

Every planar graph is 4-colorable

(Appel & Haken, 1976)

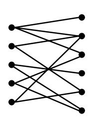
Bipartite Graphs

- A directed or undirected graph is bipartite if the vertices can be partitioned into two sets such that all edges go between the two sets

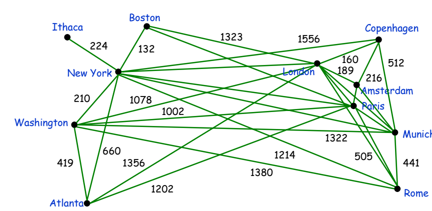


Bipartite Graphs

- The following are equivalent
  - *G* is bipartite
  - *G* is 2-colorable
  - *G* has no cycles of odd length



Traveling Salesperson



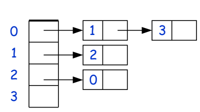
- Find a path of minimum distance that visits every city

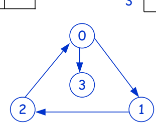
Implementing Digraphs

- Adjacency Matrix
  g[u][v] is true iff there is an edge from u to v

	0	1	2	3
0		T		T
1			T	
2	T			
3				

- Adjacency List
  The list for u contains v iff there is an edge from u to v



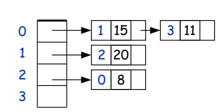


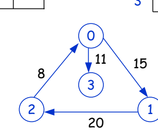
Implementing Weighted Digraphs

- Adjacency Matrix
  g[u][v] is c iff there is an edge of cost c from u to v

	0	1	2	3
0		15		11
1			20	
2	8			
3				

- Adjacency List
  The list for u contains v,c iff there is an edge from u to v that has cost c



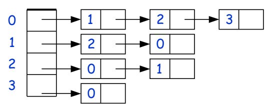


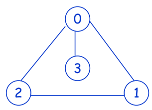
Implementing Undirected Graphs

- Adjacency Matrix
  g[u][v] is true iff there is an edge from u to v

	0	1	2	3
0		T	T	T
1	T		T	
2	T	T		
3	T			

- Adjacency List
  The list for u contains v iff there is an edge from u to v





Adjacency Matrix or Adjacency List?

n = number of vertices
m = number of edges
m_u = number of edges leaving u

- Adjacency Matrix
  - Uses space $O(n^2)$
  - Can iterate over all edges in time $O(n^2)$
  - Can answer "Is there an edge from u to v?" in O(1) time
  - Better for dense (i.e., lots of edges) graphs
- Adjacency List
  - Uses space O(m+n)
  - Can iterate over all edges in time O(m+n)
  - Can answer "Is there an edge from u to v?" in $O(m_u)$ time
  - Better for sparse (i.e., fewer edges) graphs