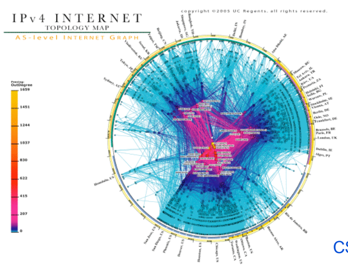


Graphs

Lecture 20
CS211 – Summer 2007

Announcements

- Prelim regrade requests due today
  - By end of consulting hours
- A3 regrade requests due tomorrow 11:59pm
- A4 due Friday 11:59pm

Quick review of recent topics…

- Asymptotic complexity
- Searching and sorting
- Basic ADTs
  - stacks
  - queues
  - sets
  - dictionaries
  - priority queues
- Basic data structures used to implement these ADTs
  - arrays
  - linked listshash 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

Quick review of recent topics…

- Language features
  - inheritance
  - inner classes
  - anonymous inner classes
  - types & subtypes
  - iteration & iterators
- GUI statics
  - layout managers
  - components
  - containers
- GUI dynamics
  - events
  - listeners
  - adapters

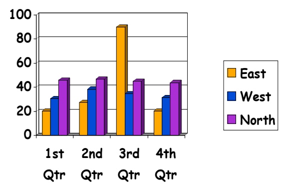
Data Structure Runtime Summary

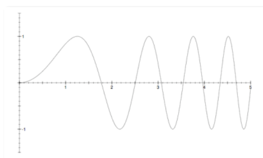
- Stack [ops = put & get]
  - O(1) worst-case time
    - Array (but can overflow)
    - Linked list
  - O(1) time/operation
    - Array with doubling
- Queue [ops = put & get]
  - O(1) worst-case time
    - Array (but can overflow)
    - Linked list (need to keep track of both head & last)
  - O(1) time/operation
    - 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) time/operation
    - 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)

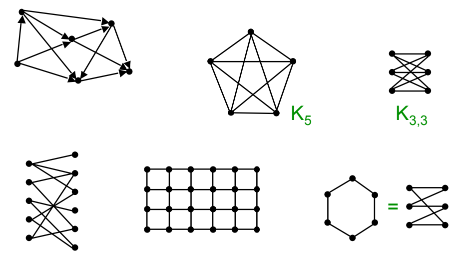
These are not Graphs





...not the kind we mean, anyway

These are Graphs



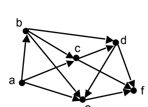
Applications of Graphs

- 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 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 (Digraph)



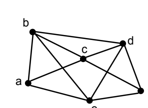
V = {a,b,c,d,e,f}
E = {(a,b), (a,c), (a,e), (b,c), (b,d), (b,e), (c,d), (c,f), (d,e), (d,f), (e,f)}

|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}

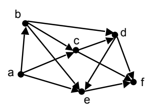
Example:

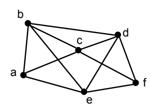


V = {a,b,c,d,e,f}
E = {{a,b}, {a,c}, {a,e}, {b,c}, {b,d}, {b,e}, {c,d}, {c,f}, {d,e}, {d,f}, {e,f}}

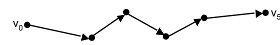
Some Graph Terminology

- 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

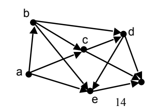




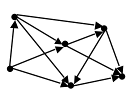
More Graph Terminology



- 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

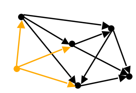


Is This a Dag?



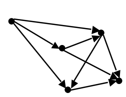
- Intuition:
  - If it's a dag, there must be a vertex with indegree zero – why?
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Is this a dag?



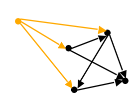
- Intuition:
  - If it's a dag, there must be a vertex with indegree zero – why?
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Is this a dag?



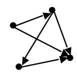
- Intuition:
  - If it's a dag, there must be a vertex with indegree zero – why?
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Is this a dag?



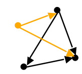
- Intuition:
  - If it's a dag, there must be a vertex with indegree zero – why?
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Is this a dag?



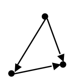
- Intuition:
  - If it's a dag, there must be a vertex with indegree zero – why?
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Is this a dag?



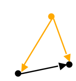
- Intuition:
  - If it's a dag, there must be a vertex with indegree zero – why?
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Is this a dag?



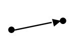
- Intuition:
  - If it's a dag, there must be a vertex with indegree zero – why?
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Is this a dag?



- Intuition:
  - If it's a dag, there must be a vertex with indegree zero – why?
- 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

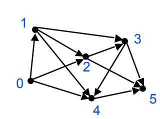
Is this a dag?



- Intuition:
  - If it's a dag, there must be a vertex with indegree zero – why?
- 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

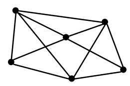
- We just computed a topological sort of the dag
  - This is a numbering of the vertices such that all edges go from lower- to higher-numbered vertices



Useful in job scheduling with precedence constraints

Graph Coloring

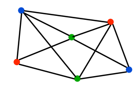
- A coloring of an undirected graph is an assignment of a color to each node such that no two adjacent vertices get the same color



- How many colors are needed to color this graph?

Graph Coloring

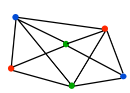
- A coloring of an undirected graph is an assignment of a color to each node such that no two adjacent vertices get the same color



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

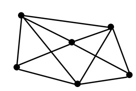
An Application of Coloring

- 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

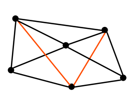
- A graph is planar if it can be embedded in the plane with no edges crossing



- Is this graph planar?

Planarity

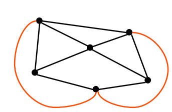
- A graph is planar if it can be embedded in the plane with no edges crossing



- Is this graph planar?
  - Yes

Planarity

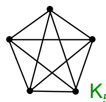
- A graph is planar if it can be embedded in the plane with no edges crossing



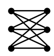
- Is this graph planar?
  - Yes

Detecting Planarity

Kuratowski's Theorem



K_5



$K_{3,3}$

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)

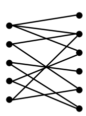
The Four-Color Theorem

Every planar graph is 4-colorable

(Appel & Haken, 1976)

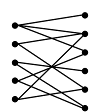
Bipartite Graphs

- 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

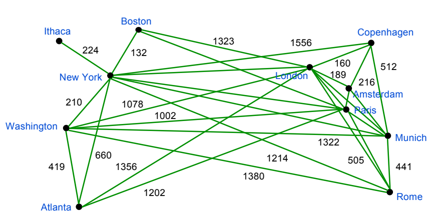


Bipartite Graphs

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

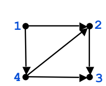


Traveling Salesperson

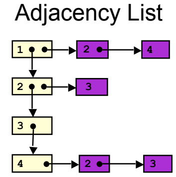


- Find a path of minimum distance that visits every city

Representations of Graphs



Adjacency List



Adjacency Matrix

	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	0
4	0	1	1	0

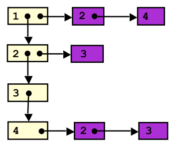
Adjacency Matrix or Adjacency List?

n = number of vertices
m = number of edges
d(u) = degree of u = number of edges leaving u

- Adjacency Matrix
  - Uses space $O(n^2)$
  - Iterate over all edges in $O(n^2)$
  - Can answer "Is there an edge from u to v?" in O(1) time
  - Better for dense graphs

	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	0
4	0	1	1	0

- Adjacency List
  - Uses space O(m+n)
  - Iterate over all edges in O(m+n)
  - Can answer "Is there an edge from u to v?" in O(d(u)) time
  - Better for sparse graphs



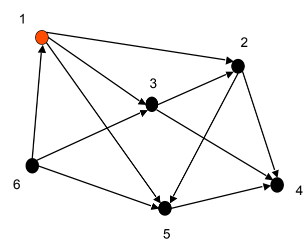
Graph Algorithms

- Search
  - depth-first search
  - breadth-first search
- Shortest paths
  - Dijkstra's algorithm
- Minimum spanning trees
  - Prim's algorithm
  - Kruskal's algorithm

Depth-First Search

- Follow edges depth-first starting from an arbitrary vertex r, using a stack to remember where you came from
- When you encounter a vertex previously visited, or there are no outgoing edges, retreat and try another path
- Eventually visit all vertices reachable from r
- If there are still unvisited vertices, repeat
- O(m) time

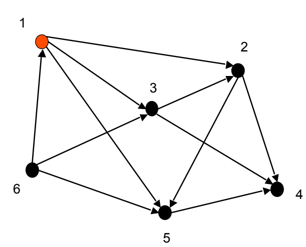
Depth-First Search



Breadth-First Search

- Same, except use a queue instead of a stack to determine which edge to explore next

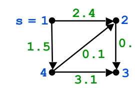
Breadth-first Search



Shortest Paths

- Suppose you have an airline route map with intercity distances. You want to know the shortest distance from Ithaca to every city on the map.
- This is the *single-source shortest path* problem.
- Can be solved using Dijkstra's algorithm
  - Assumes that edge weights are non-negative

Shortest Paths



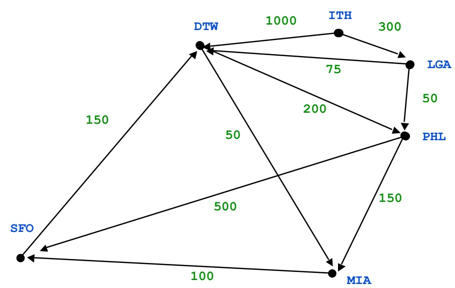
Digraph with edge weights

	1	2	3	4
1	0	2.4	∞	1.5
2	∞	0	0.9	∞
3	∞	∞	0	∞
4	∞	0.1	3.1	0

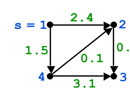
Corresponding matrix

Single-source shortest path problem: Given a graph with edge weights w(u,v) and a designated vertex s, find the shortest path from s to every other vertex (length of a path = sum of edge weights)

Shortest Paths

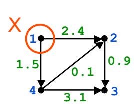


Shortest Paths



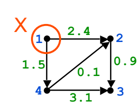
- Let d(s,u) denote the distance (length of shortest path) from s to u. In this example,
  - d(1,1) = 0
  - d(1,2) = 1.6
  - d(1,3) = 2.5
  - d(1,4) = 1.5

Dijkstra's Algorithm



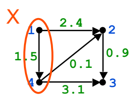
- Let X = {s}
  - X is the set of nodes for which we have already determined the shortest path
- For each node u ∉ X, define D(u) = w(s,u)
  - D(2) = 2.4
  - D(3) = ∞
  - D(4) = 1.5

Dijkstra's Algorithm



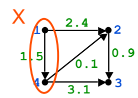
- Find u ∉ X such that D(u) is minimum, add it to X
  - at that point, d(s,u) = D(u)
- For each node v ∉ X such that (u,v) ∈ E, if D(u) + w(u,v) < D(v), set D(v) = D(u) + w(u,v)
  - D(2) = 2.4
  - D(3) = ∞
  - D(4) = 1.5

Dijkstra's Algorithm



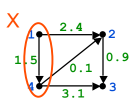
- Find u ∉ X such that D(u) is minimum, add it to X
  - at that point, d(s,u) = D(u) u = 4
- For each node v ∉ X such that (u,v) ∈ E,
  if D(u) + w(u,v) < D(v), set D(v) = D(u) + w(u,v)
  - D(2) = 2.4
  - D(3) = ∞
  - D(4) = 1.5 = d(1,4)

Dijkstra's Algorithm



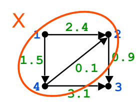
- Find u ∉ X such that D(u) is minimum, add it to X
  - at that point, d(s,u) = D(u) u = 4
- For each node v ∉ X such that (u,v) ∈ E,
  if D(u) + w(u,v) < D(v), set D(v) = D(u) + w(u,v)
  - D(2) = ~~2.4~~ 1.6
  - D(3) = ~~∞~~ 4.6
  - D(4) = 1.5 = d(1,4)

Dijkstra's Algorithm



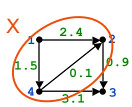
- Find u ∉ X such that D(u) is minimum, add it to X
  - at that point, d(s,u) = D(u)
- For each node v ∉ X such that (u,v) ∈ E,
  if D(u) + w(u,v) < D(v), set D(v) = D(u) + w(u,v)
  - D(2) = ~~2.4~~ 1.6
  - D(3) = ~~∞~~ 4.6
  - D(4) = 1.5 = d(1,4)

Dijkstra's Algorithm



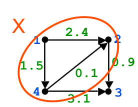
- Find u ∉ X such that D(u) is minimum, add it to X
  - at that point, d(s,u) = D(u) u = 2
- For each node v ∉ X such that (u,v) ∈ E,
  if D(u) + w(u,v) < D(v), set D(v) = D(u) + w(u,v)
  - D(2) = ~~2.4~~ 1.6 = d(1,2)
  - D(3) = ~~∞~~ 4.6
  - D(4) = 1.5 = d(1,4)

Dijkstra's Algorithm



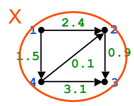
- Find u ∉ X such that D(u) is minimum, add it to X
  - at that point, d(s,u) = D(u) u = 2
- For each node v ∉ X such that (u,v) ∈ E,
  if D(u) + w(u,v) < D(v), set D(v) = D(u) + w(u,v)
  - D(2) = ~~2.4~~ 1.6 = d(1,2)
  - D(3) = ~~∞~~ ~~4.6~~ 2.5
  - D(4) = 1.5 = d(1,4)

Dijkstra's Algorithm



- Find u ∉ X such that D(u) is minimum, add it to X
  - at that point, d(s,u) = D(u)
- For each node v ∉ X such that (u,v) ∈ E,
  if D(u) + w(u,v) < D(v), set D(v) = D(u) + w(u,v)
  - D(2) = ~~2.4~~ 1.6 = d(1,2)
  - D(3) = ~~∞~~ ~~4.6~~ 2.5
  - D(4) = 1.5 = d(1,4)

Dijkstra's Algorithm



- Find u ∉ X such that D(u) is minimum, add it to X
  - at that point, d(s,u) = D(u) u = 3
- For each node v ∉ X such that (u,v) ∈ E, if D(u) + w(u,v) < D(v), set D(v) = D(u) + w(u,v)
  - D(2) = ~~2.4~~ 1.6 = d(1,2)
  - D(3) = ~~∞~~ ~~4.6~~ 2.5 = d(1,3)
  - D(4) = 1.5 = d(1,4)

Dijkstra's Algorithm

Proof of correctness – show that the following are invariants of the loop:

- For u ∈ X, D(u) = d(s,u)
- For u ∈ X and v ∉ X, d(s,u) ≤ d(s,v)
- For all u, D(u) is the length of the shortest path from s to u such that all nodes on the path (except possibly u) are in X

Implementation:

- Use a priority queue for the nodes not yet taken – priority is D(u)

Complexity

- Every edge is examined once when its source is taken into X
- A vertex may be placed in the priority queue multiple times, but at most once for each incoming edge
- Number of insertions and deletions into priority queue = m + 1, where m = |E|
- Total complexity = O(m log m)

Conclusion

- There are faster but much more complicated algorithms for single-source, shortest-path problem that run in time O(n log n + m) using something called *Fibonacci heaps*
- It is important that all edge weights be nonnegative – Dijkstra's algorithm does not work otherwise, we need a more complicated algorithm called *Warshall's algorithm*
- Learn about this and more in CS482