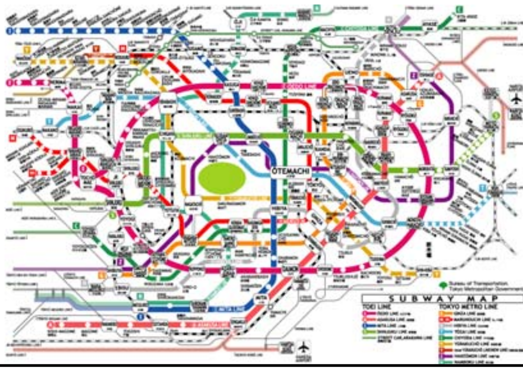
More Graphs

Lecture 21
CS211 – Fall 2005



Announcements

- Upcoming talk
  - "The Many Careers of a Computer Scientist"
    - Or how a Computer Science degree empowers you to do much more than code
  - Dan Huttenlocher, Professor in the Department of Computer Science and Johnson Graduate School of Management
  - 5:00 PM, Wednesday, November 9th
  - Upson Lounge
  - FREE PIZZA!
- ACSU (Association of Computer Science Undergraduates)

Prelim 2 Reminder

- Prelim 2
  - Tuesday, Nov 15, 7:30-9pm
  - One week from today!
  - Topics: all material through Nov 1
  - Does *not* include
    - Graphs
    - GUIs in Java
- Note that this week's Section meetings are last before the exam
- Exam conflicts
  - Email Kelly Patwell (ASAP)
- Prelim 2 Review Session
  - Sunday, Nov 13,1:30-3:00pm, Kimball B11
  - See *Exams* on course website for more 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

Implementing Digraphs

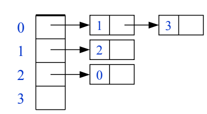
- Adjacency Matrix

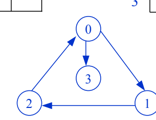
  g[u][v] is true iff there is an edge from u to v

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

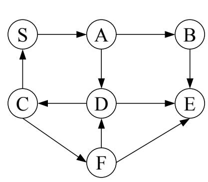
- Adjacency List

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





Shortest Paths for Unweighted Graphs




```
bfsDistance(s):
    // s is the start vertex
    // dist[v] is length of s-to-v path
    // Initially dist[v] = ∞ for all v
    dist[s] = 0;
    Q.insert(s);

    while (Q nonempty) {
        v = Q.get();
        for (each w adjacent to v) {
            if (dist[w] == ∞) {
                dist[w] = dist[v]+1;
                Q.insert(w);
            }
        }
    }
```

Analysis for bfsDistance

- How many times can a vertex be placed in the queue?
- How much time for the for-loop?
  - Depends on representation
    - Adjacency Matrix: O(n)
    - Adjacency List: $O(m_v)$
- Time:
  - $O(n^2)$ for adj matrix
  - O(m+n) for adj list

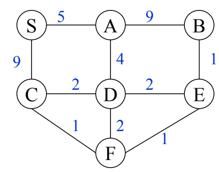
```
bfsDistance(s):
    // s is the start vertex
    // dist[v] is length of s-to-v path
    // Initially dist[v] = ∞ for all v
    dist[s] = 0;
    Q.insert(s);

    while (Q nonempty) {
        v = Q.get();
        for (each w adjacent to v) {
            if (dist[w] == ∞) {
                dist[w] = dist[v]+1;
                Q.insert(w);
            }
        }
    }
```

If There are Edge Costs?

- Idea #1
  - Add false nodes so that all edge costs are 1
  - But what if edge costs are large?
  - What if the costs aren't integers?
- Idea #2
  - Nothing "interesting" happens at the false nodes
    - Can't we just jump ahead to the next "real" node
  - Rule: always do the closest (real) node first
  - Use the array dist[ ] to
    - Report answers
    - *Keep track of what to do next*

Dijkstra's Algorithm

- Intuition
  - Edges are threads; vertices are beads
  - Pick up at s; mark each node as it leave the table
- Note: Negative edge-costs are *not allowed*

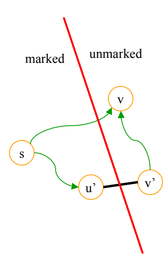


- s is the start vertex
- c(i,j) is the cost from i to j
- Initially, vertices are unmarked
- dist[v] is length of s-to-v path
- Initially, dist[v] = ∞, for all v

```
dijsktra(s):
    dist[s] = 0;
    while (some vertices are unmarked) {
        v = unmarked vertex with
            smallest dist;
        Mark v;
        for (each w adj to v) {
            dist[w] = min
                ( dist[w], dist[v] + c(v,w) );
        }
    }
```

Proof for Dijkstra's Algorithm

- Claim: When vertex v is marked, dist[v] is the length of the shortest path from s to v



- Proof
  - Suppose there is a shorter path P from s to v
  - Consider the first edge of P that links a marked vertex to an unmarked vertex
    - Such an edge must exist because we know s is marked and v is not
    - Call this edge (u',v')
  - Note that the length of the path from s to u' to v' is less than the length of P
    - Thus v' would be chosen in the algorithm instead of v
    - Contradiction!

Dijkstra's Algorithm using Adj Matrix

- While-loop is done n times
- Within the loop
  - Choosing v takes O(n) time
    - Could do this faster using PQ, but no reason to
  - For-loop takes O(n) time
- Total time = O(n^2)

- s is the start vertex
- c(i,j) is the cost from i to j
- Initially, vertices are unmarked
- dist[v] is length of s-to-v path
- Initially, dist[v] = ∞, for all v

```
dijsktra(s):
    dist[s] = 0;
    while (some vertices are unmarked) {
        v = unmarked vertex with
            smallest dist;
        Mark v;
        for (each w adj to v) {
            dist[w] = min
                ( dist[w], dist[v] + c(v,w) );
        }
    }
```

Dijkstra's Algorithm using Adj List

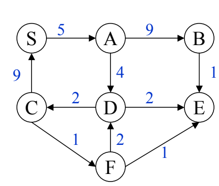
- Looks like we need a PQ
  - Problem: priorities are updated as algorithm runs
  - Can insert pair (v,dist[v]) in PQ whenever dist[v] is updated
  - At most m things in PQ
- Time O(n + m log m)
- Using a more complicated PQ (e.g., Pairing Heap), time can be brought down to O(m + n log n)

- s is the start vertex
- c(i,j) is the cost from i to j
- Initially, vertices are unmarked
- dist[v] is length of s-to-v path
- Initially, dist[v] = ∞, for all v

```
dijsktra(s):
    dist[s] = 0;
    while (some vertices are unmarked) {
        v = unmarked vertex with
            smallest dist;
        Mark v;
        for (each w adj to v) {
            dist[w] = min
                ( dist[w], dist[v] + c(v,w) );
        }
    }
```

Dijkstra's Algorithm for Digraphs

- Algorithm works on both undirected and directed graphs without modification
- As before: Negative edge-costs are *not allowed*



- s is the start vertex
- c(i,j) is the cost from i to j
- Initially, vertices are unmarked
- dist[v] is length of s-to-v path
- Initially, dist[v] = ∞, for all v

```
dijsktra(s):
    dist[s] = 0;
    while (some vertices are unmarked) {
        v = unmarked vertex with
            smallest dist;
        Mark v;
        for (each w adj to v) {
            dist[w] = min
                ( dist[w], dist[v] + c(v,w) );
        }
    }
```

Greedy Algorithms

- Dijkstra's Algorithm is an example of a *Greedy Algorithm*
- The Greedy Strategy is an algorithm design technique
  - Like Divide & Conquer
- The Greedy Strategy is used to solve optimization problems
  - The goal is to find the best solution
- Works when the problem has the *greedy-choice property*
  - A global optimum can be reached by making locally optimum choices
- Problem: Given an amount of money, find the smallest number of coins to make that amount
- Solution: Use a Greedy Algorithm
  - Give as many large coins as you can
- This greedy strategy produces the optimum number of coins for the US coin system
- Different money system ⇒ greedy strategy may fail
  - For example: suppose the US introduces a 4¢ coin

Minimum Spanning Trees

Definition

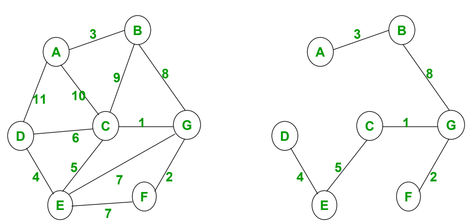
A *spanning tree* of an undirected graph G is a *tree* whose nodes are the vertices of G and whose edges are a subset of the edges of G

Definition

A *Minimum Spanning Tree* (*MST*) for a weighted graph G is the spanning tree of least cost (sum of edge-weights)

- Alternately, an MST can be defined as the least-cost set of edges so that all the vertices are connected
  - This has to be a tree… Why?
- A greedy strategy works for this problem
  - Add vertices one at a time
  - Always add the one that is closest to the current tree
  - This is called *Prim's Algorithm*

An Example Graph and Its MST



Prim's Algorithm

- s is the start vertex
- c(i,j) is the cost from i to j
- Initially, vertices are unmarked
- dist[v] is length of smallest tree-to-v edge
- Initially, dist[v] = ∞, for all v

```
prim(s):
    dist[s] = 0;
    while (some vertices are unmarked) {
        v = unmarked vertex with
            smallest dist;
        Mark v;
        for (each w adj to v) {
            dist[w] = min[ dist[w], c(v,w) ];
        }
    }
```

- Runtime analysis
  - $O(v^2)$ for adj matrix
    - While-loop is executed v times
    - For-loop takes O(v) time
  - O(e + v log v) for adj list
    - Use a PQ
    - Regular PQ produces time O(v + e log e)
    - Can improve to O(e + v log v) by using fancier heap

Similar Code Structures

```
while (some vertices are unmarked) {
    v = best of unmarked vertices;
    Mark v;
    for (each w adj to v)
        Update w;
}
```

- bfsDistance
  - best: next in queue
  - update: dist[w] = dist[v]+1
- dijkstra
  - best: next in PQ
  - update:dist[w] =min [dist[w],dist[v]+cost(v,w)]
- prim
  - best: next in PQ
  - update: dist[w] = min [dist[w],cost(v,w)]

Remembering Your Choices

- How can you remember which choices were made?
  - Whenever dist[w] is updated we can remember the current v by using parent[w] = v;
  - Can use the parent info to construct the *bfs tree*, the *shortest path tree*, or the *minimum spanning tree*

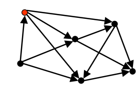
```
while (some vertices are unmarked) {
    v = best of unmarked vertices;
    Mark v;
    for (each w adj to v)
        Update w;
        if (w changed) parent[w] = v;
}
```

Depth-First Search

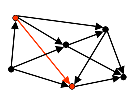
- Follow edges depth-first starting from an arbitrary vertex s, using a *Stack* to remember where you came from
- When you encounter a vertex previously visited, or there are no outgoing edges, retreat and try another path
- Eventually visit all vertices reachable from s
- If there are still unvisited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

Easy to see this takes O(m) time

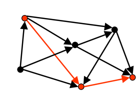
Depth-First Search



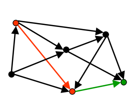
Depth-First Search



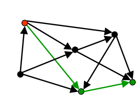
Depth-First Search



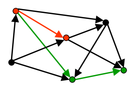
Depth-First Search



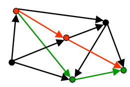
Depth-First Search



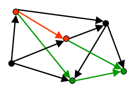
Depth-First Search



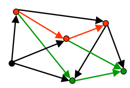
Depth-First Search



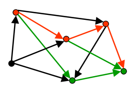
Depth-First Search



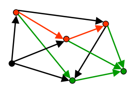
Depth-First Search



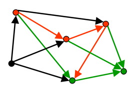
Depth-First Search



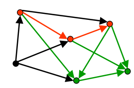
Depth-First Search



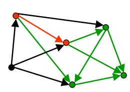
Depth-First Search



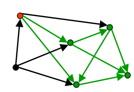
Depth-First Search



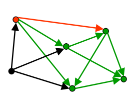
Depth-First Search



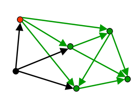
Depth-First Search



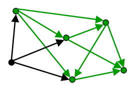
Depth-First Search



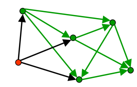
Depth-First Search



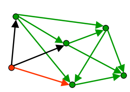
Depth-First Search



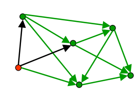
Depth-First Search



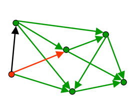
Depth-First Search



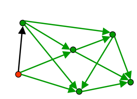
Depth-First Search



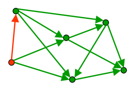
Depth-First Search



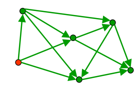
Depth-First Search



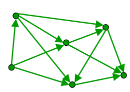
Depth-First Search



Depth-First Search



Depth-First Search



DFS Notes

- Same as BFS, except we use a Stack instead of a Queue to determine which edge to explore next
- Can also implement DFS recursively
  - The Stack is represented *implicitly* in the Stack Frames created by the recursive calls