Graph Algorithms

Nick Galbreath

nickg@friendster.com

Questions

- Questions

Backtrack, Review, Corrections

- A *graph* is a set of nodes, with a set of edges.
- A *weighted* graph is a graph where each edge has a associated numerical value (think distance)
- A *undirected graph* is one where the edges are bi-directional (communative?) Edge (a,b) means at *a* you can connect to *b,* at *b* you can connect to *a*
- A *directed* graph edges are directional. Edge(*a*, *b*) means at *b* you can *not* connect to *a*
-
- *I'm focusing on undirected graphs (social networks).*

Embedded Adjacentcy Lists

- One can also add the adjacentcy list into the node itself.

```
Class Node {
    public String key;
    public Node[] neighbors;
    ...other data...
}
```

- Need a map from a key to the node to get started. Somehow need the first node.
- Good since, after you have the first node you don't need to do a map lookup (more expensive)
- More expensive in terms of memory, but only for the largest graphs is that a problem.

More on Embedded Adj Lists

- Also the Node class will need a method to add edges to itself.

```
Class Node {
    public String key;
    public Node[] neighbors;
    ...other data...
       public void addEdge(Node n);
       public void removeEdge(Node n);
}
```

  - *Your graph will also need a way of deleting a node.*
  - *Pros and Cons for each way.*

“Degree”

- Mathematically, the degree of a node is the number of adjacent vertices.
- It's the number of friends.
- It's not the “degree of separations”
- That's typically known as “distance” or “depth”

Graph Traversals

Graph Traversal

- Given a starting point, walk through every node in the graph *once.*
- Similar to a java.util.Iterator
- More than one way of doing this
- Two main methods
  - Depth-First
  - Breadth-First

Depth First

- Depth-First traversals are similar to a letting a child loose in a maze
- Run as far away from the starting point and then backtracking up one level when you run out of options.

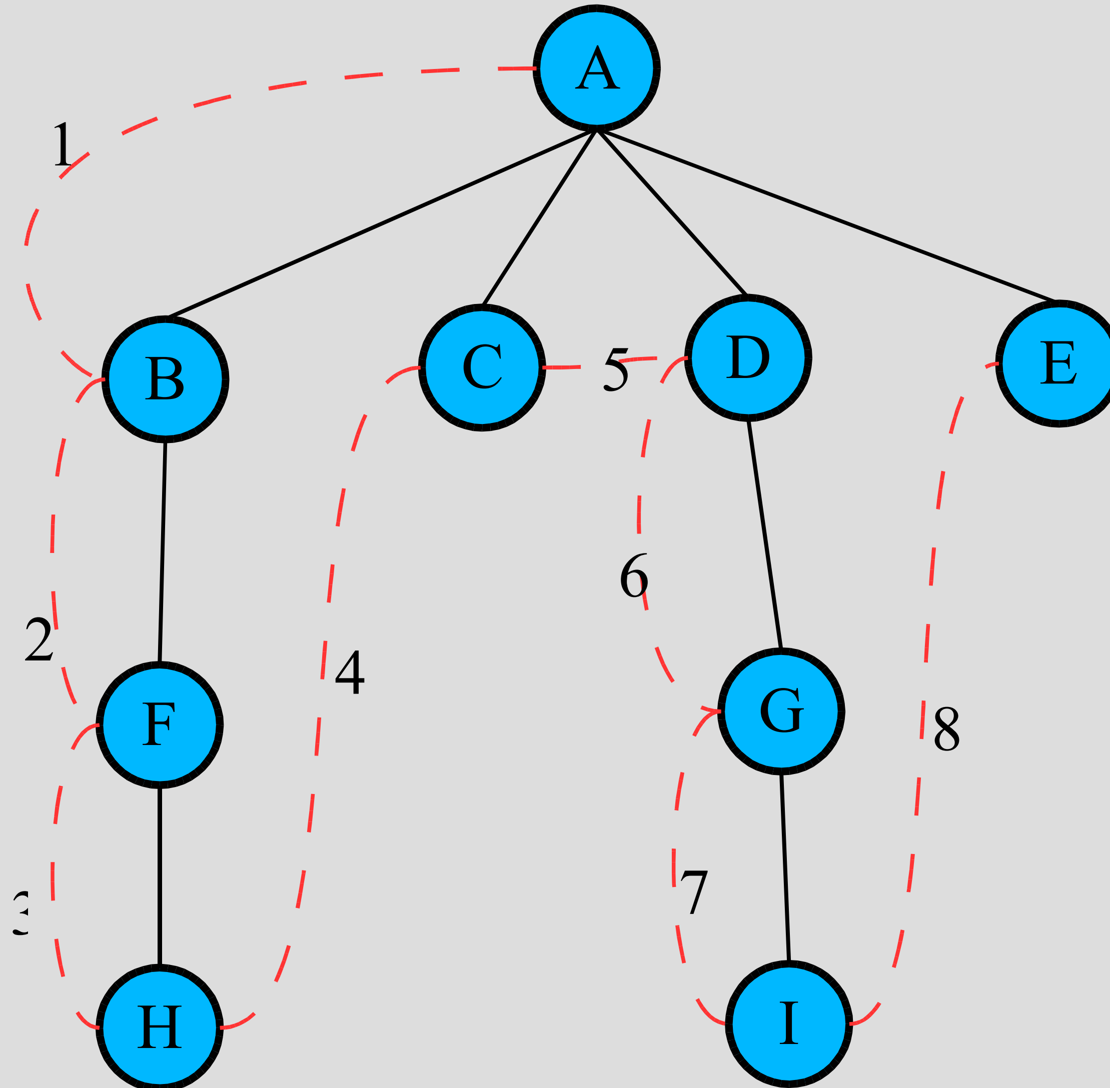
Depth-First Rules

- From the text
  - Rule 1: If possible, visit an adjacent unvisited node, mark it as visited and push on the stack
  - Rule 2: If you can't follow Rule 1, pop a vertex off the stack
  - Rule 3: If you can't follow Rule 1 or Rule 2 you are finished.

Depth First Pseudo-Code

```
stack.push(first node)
path.add(first node);
while (! stack.isEmpty()) {
    adj = graph.getAdjacent(stack.peek());
    Node v = first unvisted node in adj or null if none;
    if (v == null) {
        stack.pop(); // backup
    } else {
        stack.push(v);
        path.add(v);
    }
}  // visited contains the “path”
```

Sample Path DFS



Stacks and LinkedList

- java.util.LinkedList can be used as a stack
  - addLast == push
  - removeLast == pop
  - getLast == peek
- Why not use java.util.Stack?
  - Stack, Vector and Hashtable are kinda funny and really shouldn't be used unless you have a good reason too. (Why? They are synchronized (slow) and generally more-or-less deprecated classes.)
  - Use ArrayList/LinkedList or HashMap instead.

Checking if Visited

- The Book uses a Node class and special “ifVisited” boolean flag
- Ok, for “single-threaded” applications (desktop application)
- Not good for server applications, or applications with concurrent operations on the same graph

Checking if Visited 2

- Every time a new node is found, it must be added *somewhere* to a data structure so you can check later.
- Any of the java.util.List types (LinkedList, ArrayLists) can be used, by using the "contains()" method. (Maybe slow since it's a linear search)
- HashSet – fast look up by hashing
- TreeSet – fast look up by using a tree (sorted elements)

Depth-First?

- Depth first does not necessarily go “deep” relative to the starting node.
- It's not quite the same thing as “distance first”

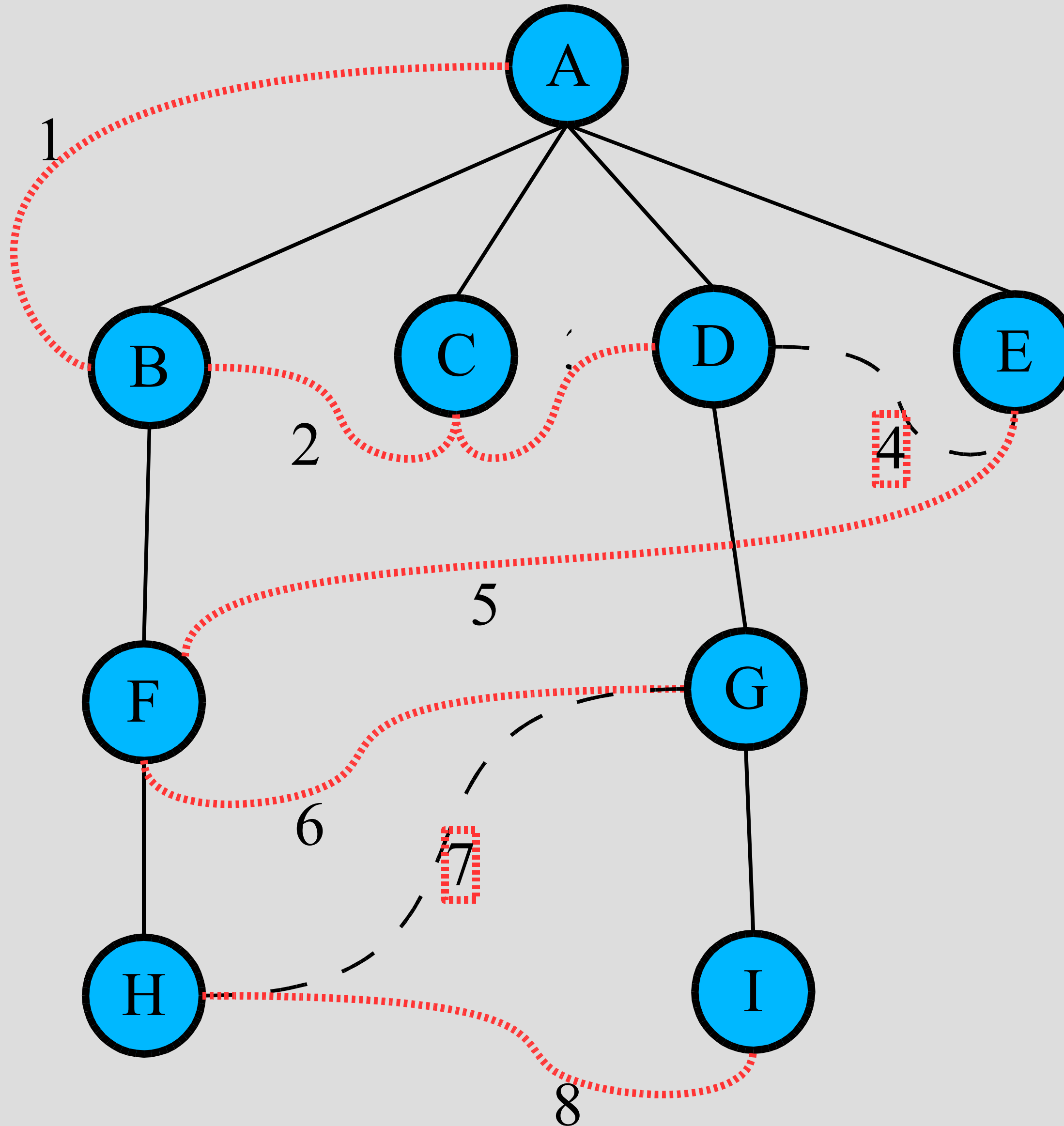
Breadth First

- Breadth-First traversals try to stay as close to the start node as possible.
- Like ripples in a wave.
-

Rules for BFS

- Rule 1: Visit the next unvisited node (if there is one) that's adjacent to the current node, mark it, and insert into the queue
- Rule 2: If rule 1 fails since all adjacent nodes are already visited, remove a node from the queue and make it the current node.
- Rule 3: If rule 2 fails since the queue is empty, then you are done.

Sample Path BFS



PsuedoCode for BFS

```
Mark top node as visited
queue.insert(top node);
path.add(top node);
while (!queue.isEmpty()) {
    Node current = queue.remove();
    Node[] adj = graph.getAdjacent(current);
    foreach node in adj
        if (node is unvisited) {
            mark node as visited
            queue.insert(node);
            path.add(node);
        }
    }
}
```

Queues and LinkedLists

- Similar to stacks, queues can also be represented using the java.util.LinkedList
- “insert” == addLast
- “remove” == removeFirst

Which Traversal to Use?

- Depends on application and on graph data
- Breadth-First is typically easier and faster to use.

Applications of Traversals

- Graph “connectedness”
- Subgraph extraction
- Minimal Spanning Trees

Graph Connectedness

- I lied. Both types of traversal will visit every node that is connected to the starting point.
- Comparing the results of traversal with the number of nodes in a graph will tell you if the graph is connected or not.
- Who cares?
- It's can be a Big Deal if you every or any node can not be reached by any others.

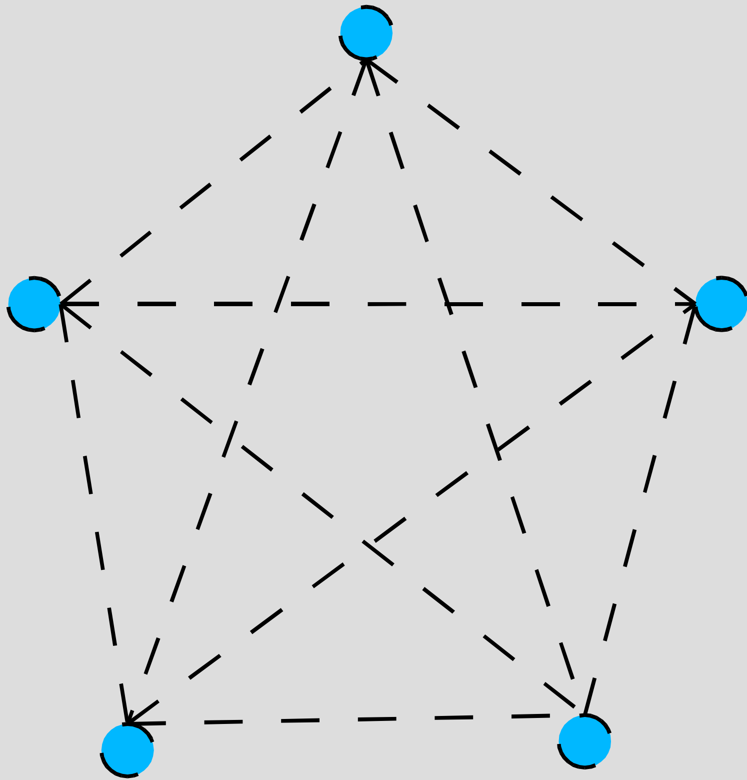
Subgraphs

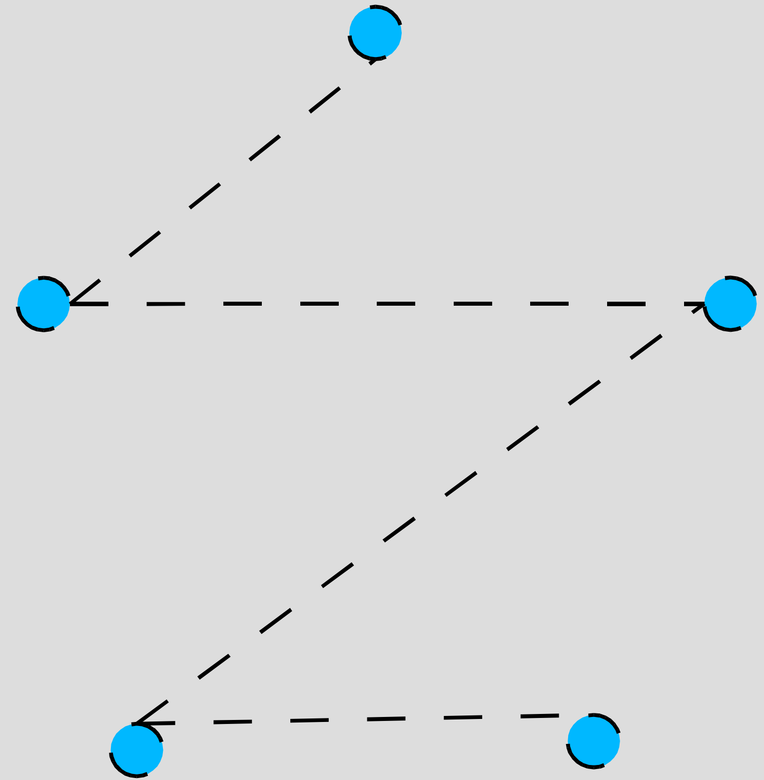
- Minor modifications can limit the traversal to a certain depth (or degree of seperation) to the starting point.
- This is your “personal network” or “network” in Friendster-speak.
-

Minimal Spanning Trees

- Uses a mimimum number of edges to connect all nodes
- Only one path between two adjacent nodes
- If you use a depth-first search the path created automatically creates a MST.
- Not Unique!
-
-
- (note: in an unweighted graph, every spanning tree is minimal)

Minimum Spanning Trees





Weighted Graphs

- Each edge has a “cost” or “distance” associated with it.
- Frequently the graphs are directed a-->b, but b --> a
- Many important algorithms includes
  - Finding the minimum cost spanning tree
  - The shortest or least costly path between two nodes
  - Your book has very detailed descriptions of all of this.

Weighted Graph Algorithms

- Many of the algorithms in the book are fine for “small” graphs
- Orbitz, Google and other places do *not* use them.
- There is a whole class of “probablistic algorithms” that do not find the optimal solution, but find the a solution that's 1% away from being optimal.
- Way beyond this class. Just an FYI that they exist.

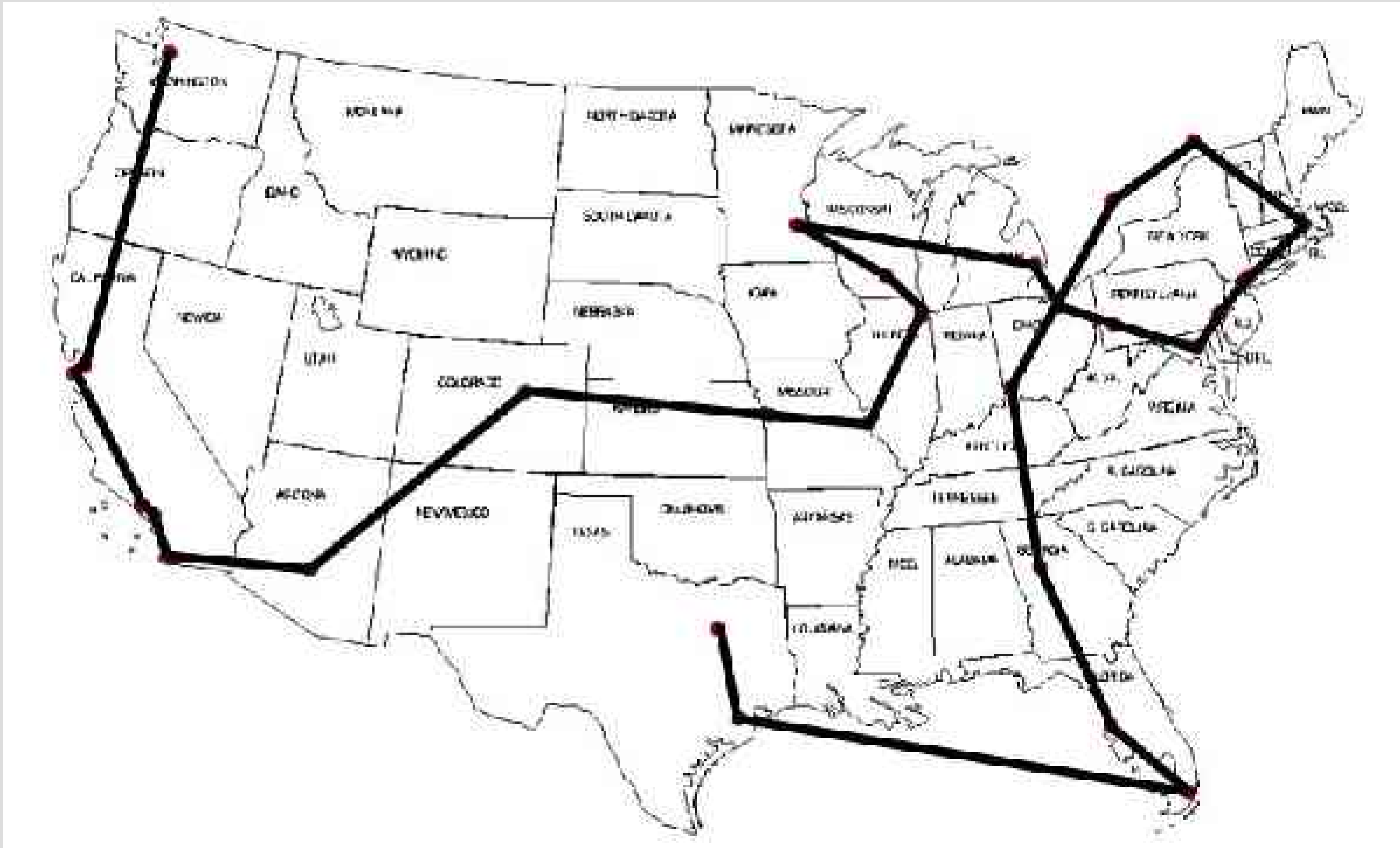
Traveling Salemen Problem

- The shortest walk on a weighted graph (i.e. Visit every node with the least cost).
- From a VW advertisment a while back

  “What is the shortest distance one needs to travel to visit all 30 teams in 28 major league cities?”
- A resonable way might be

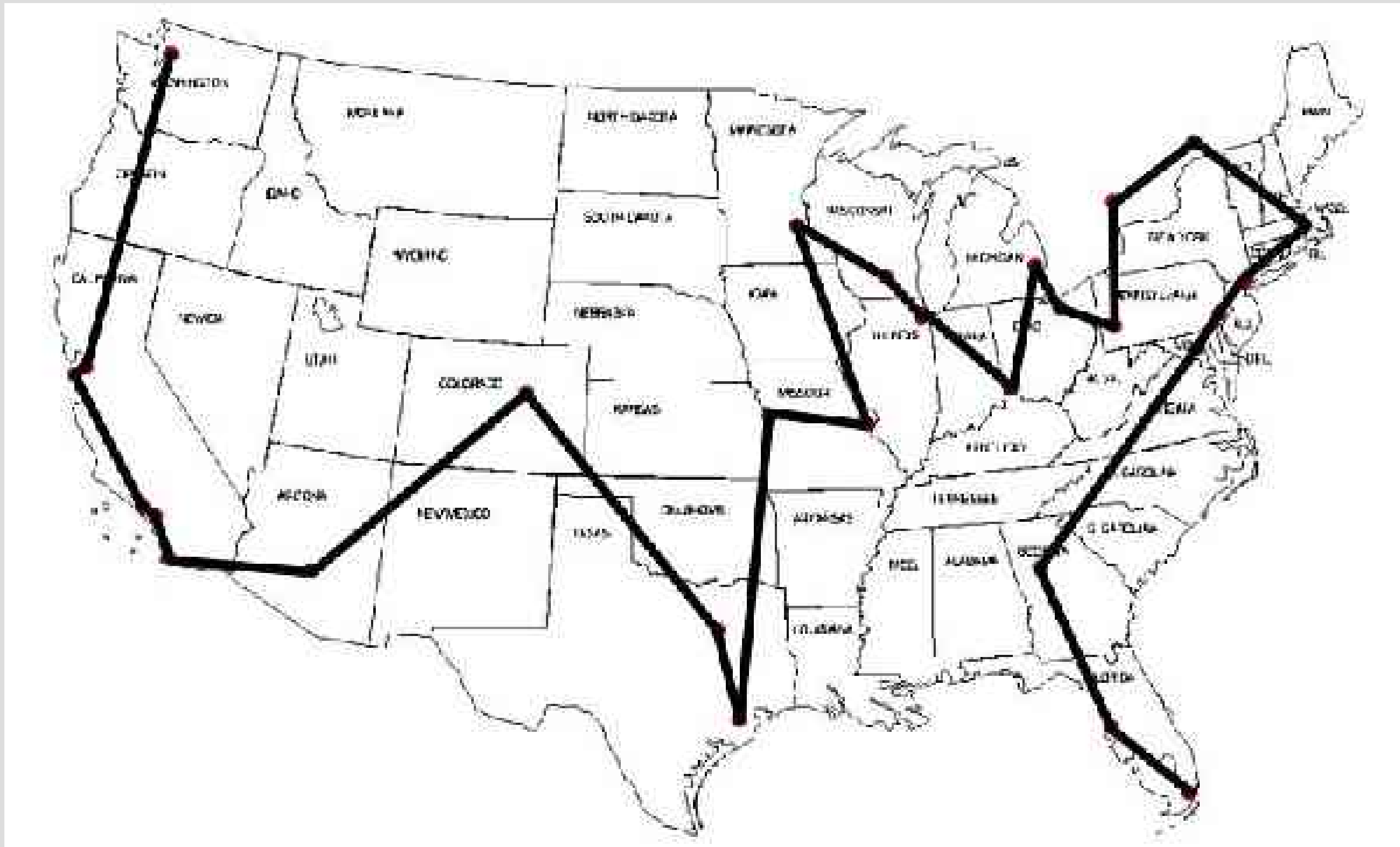
  “Starting in Seattle, go to the closest Major League city we have not visited yet. and keep doing this until we see all 28 cities.”

Traveling Saleman 2



Yuck. Certainly not the shortest path

Traveling Salesman 3



This might be the shortest path.

Traveling Salesman Conclusion

- Might be since it's proven that the only way to know for certain is to try EVERY possible path.
- This is ugly. Even for small graphs.

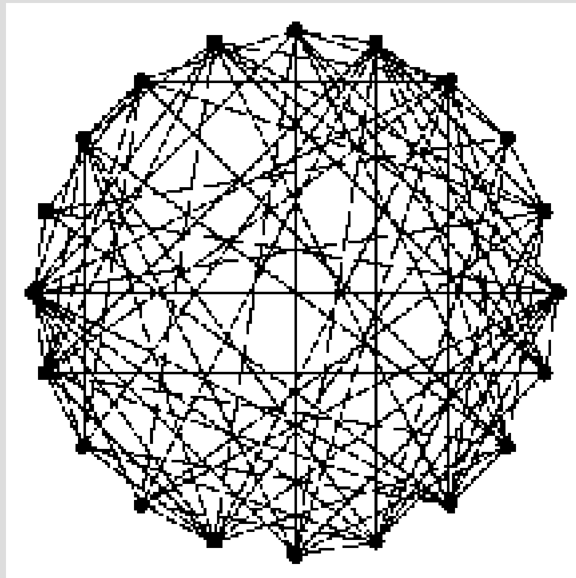
- Shameless borrowed from http://members.cox.net/mathmistakes/travel.htm

Graph Visualization

- Given an arbitrary graph, there is no canonical “visualization” -- it's not like a tree.
- Visualizations are frequently called “embeddings”
- Each is specifically designed to show off a particular property of the graph.

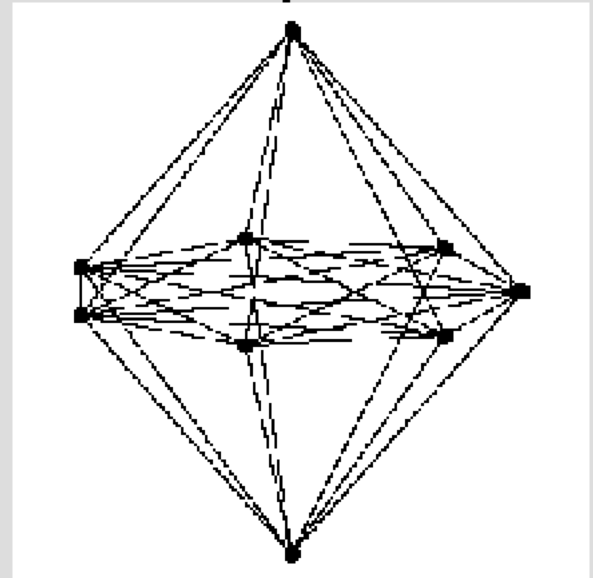
Circular Embeddings

- Shows the adjacent lists visually, but not much structure beyond that.

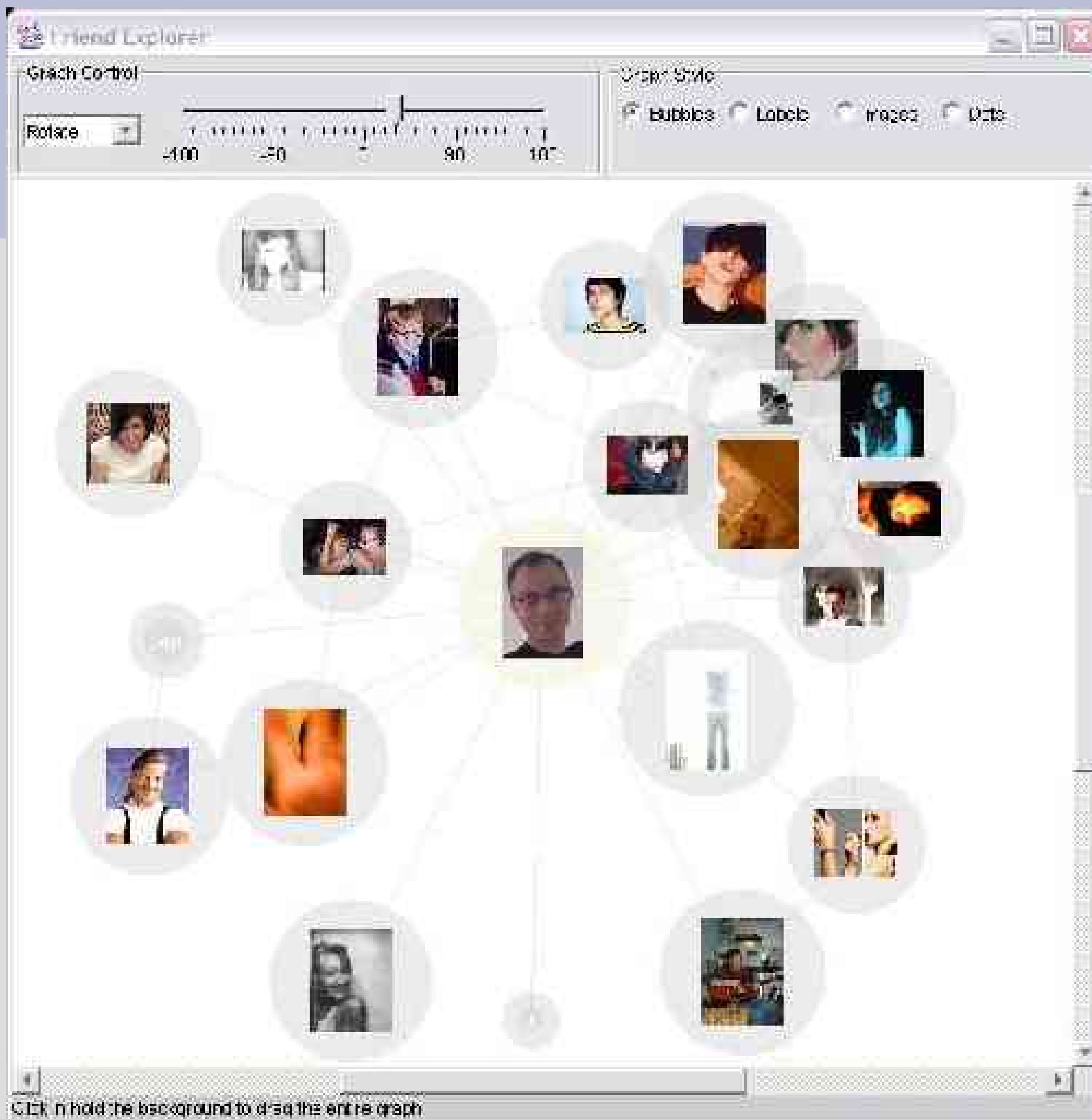


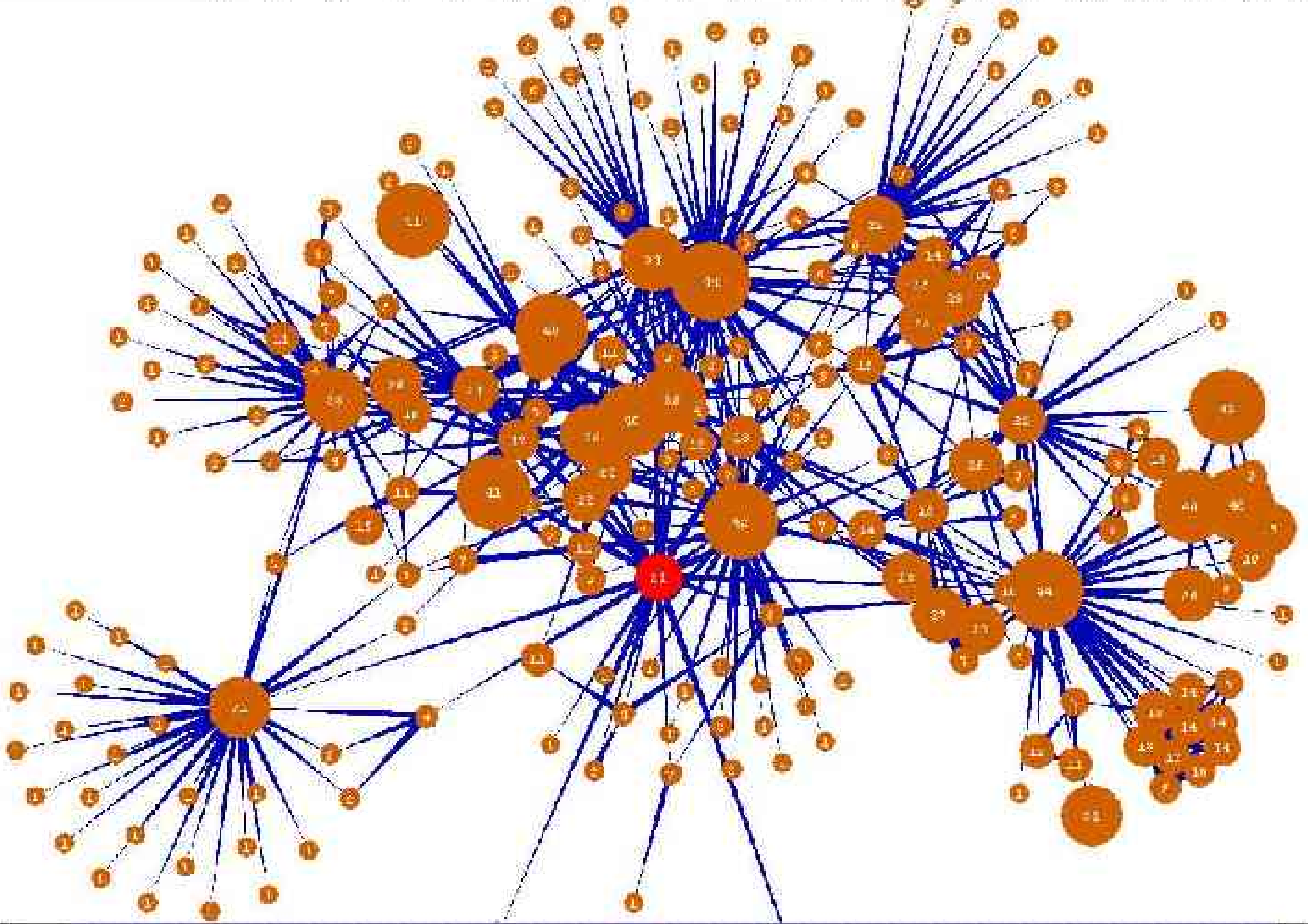
Spring Embedding

- Pretend each edge is connected by a “spring” and use Hooke's Law.
- Compute “minimal energy”
- Nodes with more connections are squished together
- Less connected nodes tend to fly away
- Slow!
- See www.touchgraph.com



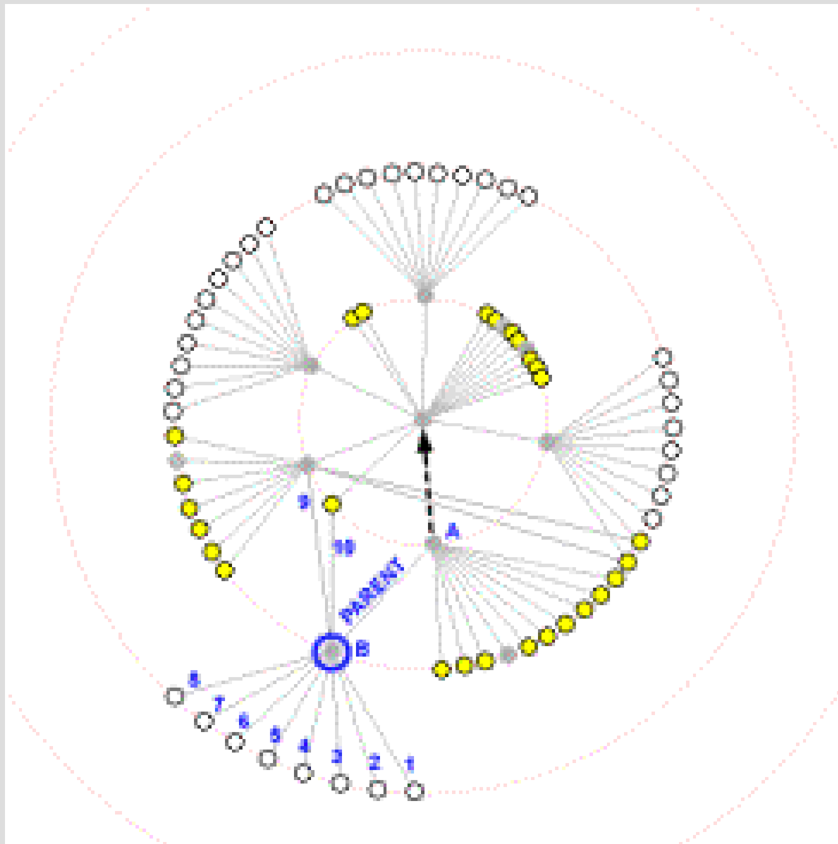
www.combinatorica.com


Graph Control
Rotate
Bubbles
Labels


Friend Explorer
Zoom
10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200

Radial Embedding

- Shows off degrees of separation



http://bailando.sims.berkeley.edu/papers/infovis01.htm

Exciting Project!

- webster.com is an online dictionary and thesaurus.
- Especially interesting is that the words link to other words – a graph!
- Explore how different words are related by slurping in their data
- Since the goal is work on graphs and not the finer points of java.io or java.util.regex here's some handy functions.

A Simple Graph Interface

```
import java.util.ArrayList;
public interface Graph {
    public getAdjacent(String key, ArrayList list)
    public ArrayList setAdjacent(String key)
    public int nodes();
    public int edges();
    public void addNode(String key);
    public void deleteNode(String key);
    public void addEdge(String key1, String key2);
    public String toString(); // dump it to text
    public int hashCode();// ??
    public boolean equals(); // ??
    }
```

Page Sucker

```
public static String getWebsterPage(String keyword) throws
  IOException {
   URL url = new URL(
       "http://www.webster.com/cgibin” +
         ”/thesaurus?book=Thesaurus&va="+ keyword);
   StringBuffer buf = new StringBuffer(5000);
   BufferedReader is =
       new BufferedReader(
       new InputStreamReader((InputStream) url.getContent()));
   String line;
   while ((line = is.readLine()) != null) {
       buf.append(line);
       buf.append('\n');
   }
   return buf.toString();
}
```

Not the best way!

Keyword Extractor

```
public static final Pattern regex =
     Pattern.compile("va=([a-zA-z]+)");

public static ArrayList getKeywordsFromPage(String page) {
    ArrayList list = new ArrayList();
    Matcher m = regex.matcher(page);
    while (m.find()) {
        // group() returns the entire match;
        // group(1) return what matched in the parens
        list.add(m.group(1));
        System.out.println(m.group(1));
    }
    return list;
}

// need to import java.util.* and java.util.regex.*;
```

Good Manners

When screwing around with a website, it's a good idea to sleep for a while between requests.

```
public static void sleepSeconds(int n) {
    try {
      Thread.sleep(n * 1000);
    } catch (Exception e) {
        // nothing
    }
}
```

Goal

- You need to write a graph implementation with Strings as the nodes.
- You need to slurp related words from webster.com using the breadth or depth-first algorithms
- You should be able to print in some manner the graph using adjacentcy lists.
- I have no idea what the result will be!