SIG Algorithm Challenges

Week 7: Graphs

Dane's Contact Email: dziema2@uic.edu
Github: Dane8373

Slides for this week available NOW at
https://github.com/dane8373/SIG_Algorithm_Challenges/tree/master/Week7

Join our Slack channel, #sig_algorithm_chal on the UIC ACM slack (uicacm.slack.com)

Mock Technical Interviews Available!

Sign up here: http://bit.ly/SIG_AC_TECH

Graph Introduction

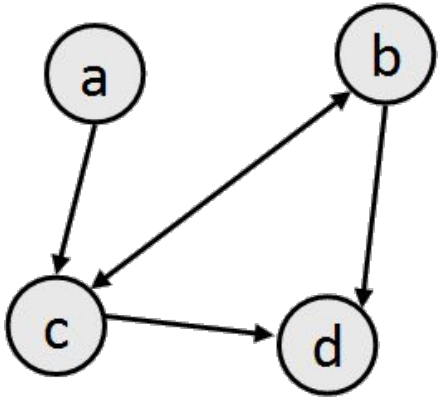
Much of a data on the internet, such as friend groups, company portfolios or product diagrams can be represented with graphs. Therefore, they can be incredibly useful

- For Computer science applications, there are three common representations of graphs, each with various pros and cons
  - Adjacency List
  - Adjacency Matrix
  - Object Collections
  - There is also a less common Edge list, but they will not be covered

- Unlike other data structures such as HashMaps, often times on questions you have to design your own type of graph, and determine the best way to take input and translate it into a graph

- Most graph problems are solved by the workhorses of graph algorithms: Breadth first search, Depth first search, and Djikstra's algorithm

Graph Types: Adjacency list



An adjacency List is a list of Vertices, where each Vertex contains a list of all vertices that it is adjacent to.

Example:
Class Vertex

String ID

List<Vertex> outgoingEdges

Class Graph

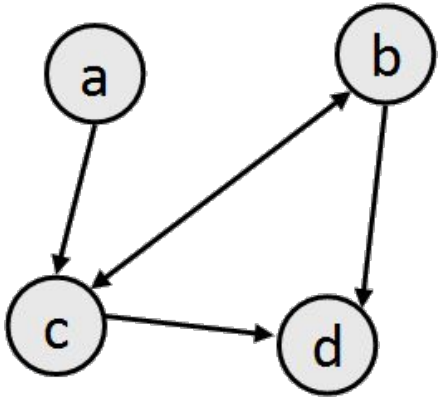
List<Vertex> Vertices

An exampe of vertex b above would be (“b”,[c,d])

The above map would be the following list. G = { (“a”,[c]), (“b”,[d,c]), (“c”,[d,a]), (“d”,[]) }]

Operation	**Time**
vertices	$O(n)$
edges	$O(m)$
endVertices, opposite	$O(1)$
incidentEdges	$O(\deg(v))$
areAdjacent	$O(\min(\deg(u), \deg(v)))$
replace	$O(1)$
insertVertex, insertEdge, removeEdge,	$O(1)$
removeVertex	$O(\deg(v))$

Graph Types: Adjacency Matrix



An adjacency matrix representing *n* vertices would be a integer matrix of size n x n. Each row represents a vertex, and each column represents potential edges between vertices. Example: if Matrix[0][1] is 1, then that means there is an edge between vertex 0 and 1

The above graph would be represented by the following matrix

0	0	1	0
0	0	1	1
0	0	0	1
0	0	0	0

This can be augmented to a weighted graph, by replacing the ones with the weight of the various edges

Operation	**Time**
vertices	$O(n)$
edges	$O(m)$
endVertices, opposite	$O(1)$
incidentEdges	$O(n + \deg(v))$
areAdjacent	$O(1)$
replace	$O(1)$
insertEdge, removeEdge	$O(1)$
insertVertex, removeVertex	$O(n^2)$

Graph Types: Object Collections

Object Collections are essentially an augmentation of an adjacency list with more information example:

Class Vertex

List<Edge> outgoingEdges

Class Edge

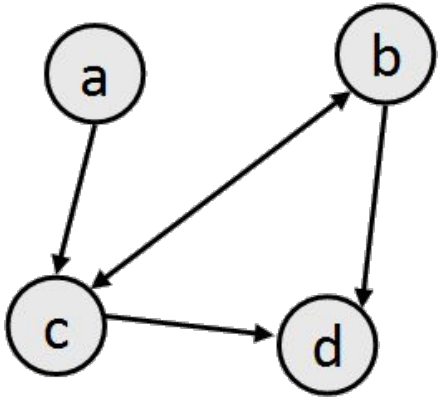
Vertex Source

Vertex Dest

Int weight

Class Graph

List<Vertex> Vertices



Graph Types: Pros and Cons

Adjacency List/Object Collections

Pros: Intuitive, easy to add/remove vertices, much lower memory requirement for sparse graphs

Cons: Not as fast as adjacency matrix, doesn't save space if graph is very dense

Good for: Graphs that change frequently, Graphs that have relatively small amount of edges

Adjacency Matrix:

Pros: Very fast to access edges, see if an edge exists or not.

Cons: Very costly to add/remove a vertex, Always costs O(n^2) space no matter how many edges exist

Good for: Dense, static graphs.

Graph Fundamentals: BFS

Breadth first search searches is an algorithm to find all reachable nodes from a given start. It searches nodes that are closest to the start, and uses a queue to accomplish this. You have to keep track of a visited set as well, using either a set or by adding a visited field to the vertex class

Pseudocode for Adjacency List BFS

Add the start vertex to Queue

While (Queue is not empty)

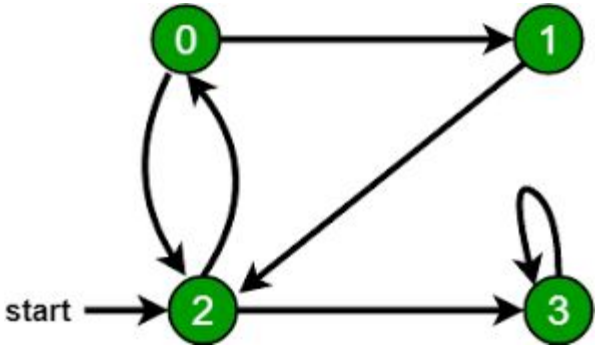
    CurrentNode = Queue.remove()

    Mark CurrentNode as visited

    For (Every Vertex V in Current Nodes Adjacency List)

        If V is not visited

            Add V to Queue



Graph Fundamentals: BFS

Pseudocode for Adjacency Matrix BFS

Add the Index start vertex to Queue

While (Queue is not empty)

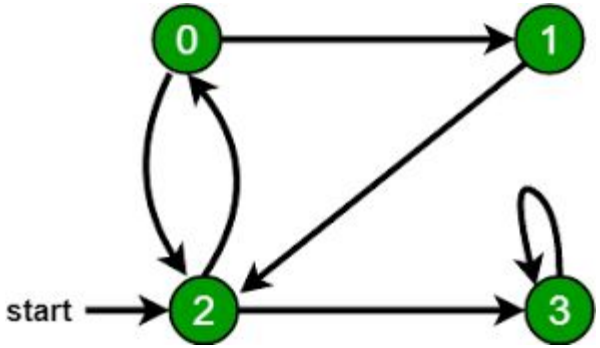
    CurrentIndex = Queue.remove()

    Mark CurrentIndex as visited

    For (Every Column Index C that isn't 0 in Row CurrentIndex of Matrix)

        If C is not visited

            Add C to Queue



Graph Fundamentals: DFS

Depth first search is similar to Breadth first search implementation wise, but it searches vertices immediately when it encounters an edge leading to that vertx. This can be implemented by either replacing the Queue in the BFS examples with a stack, or by using recursion

Pseudocode for Recursive Adjacency List DFS

DFS(v)

    If v his already visited

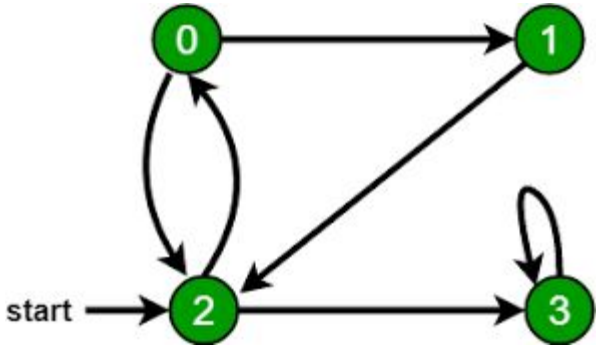
        Return

    Else

        Mark V as visited

        For (every Vertex v2 in v's adjacency list)

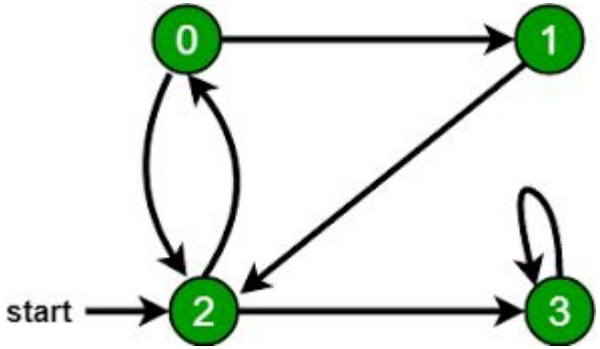
            DFS(V2)



Graph Fundamentals: DFS

Pseudocode for Adjacency Matrix DFS

```
DFS(row, Matrix)
        If row is  already visited
                Return
        Else
                Mark row as visited
                For (Every Column Index C that isn't 0  in Row CurrentIndex of Matrix)
                        DFS(C, Matrix)
```



Graph Fundamentals: Djikstras

Djikstras algorithm is a BFS for weighted graphs that returns a shortest path tree. It accomplishes this by greedily selecting the smallest edges going out of the vertices that have already been visited , and taking the one with smallest cost. It uses a priority queue, and you need to keep track of the current cost of all vertices

Pseudocode for Adjacency List Djikstras

Set all vertex costs to infinity, except set the start to 0

Add the start vertex to PriorityQueue

While (PriorityQueue is not empty)

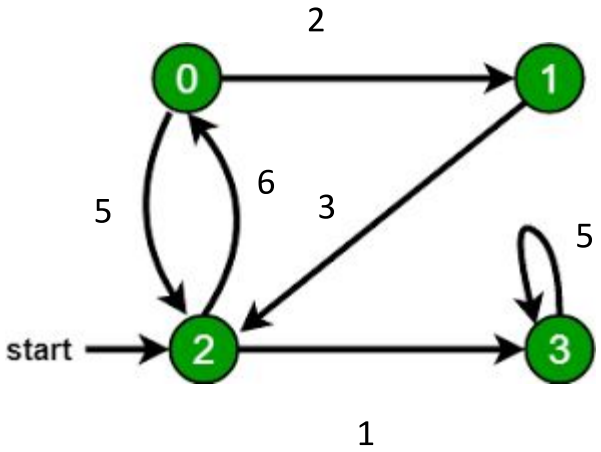
CurrentNode = PriotiyQueue.remove()

Mark CurrentNode as visited

For (Every Vertex V in Current Nodes Adjacency List)

If V is not visited and (CurrentNode's cost + the edge to V's cost) < V's current cost

Add V to PriotiyQueueand update its cost to (CurrentNode's cost + the edge to V's cost)



Graph Fundamentals: Djikstras

Pseudocode for Adjacency Matrix Djikstras

Set all vertex costs to infinity, except set the start to 0

Add the start vertex Index to Queue

While (Queue is not empty)

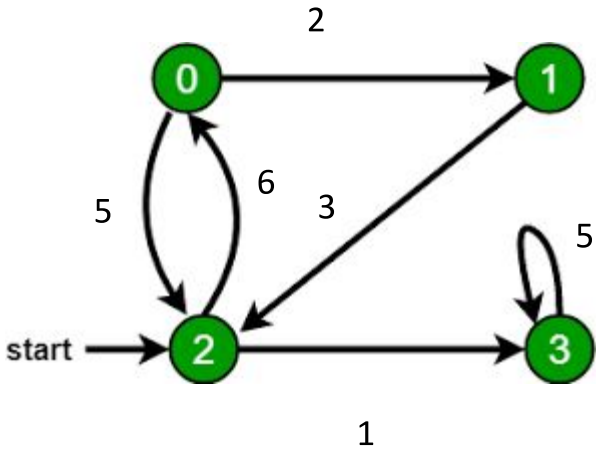
CurrentNode = Queue.remove()

Mark CurrentNode as visited

For (Every Column Index C that isn't 0 in Row CurrentIndex of Matrix)

If V is not visited and (CurrentNode's cost + the edge to V's cost) < V's current cost

Add V to Queue and update its cost to (CurrentNode's cost + the edge to V's cost)



General Tips for Using Graphs

- Some Graph problems are easy to identify, they are explicitly given as a graph.
- Other times, it isn't quite as easy, but it is often still apparent that it is a graph problem
- Any time there is some sort of spatial relationship, or some sort of relationships between multiple entites, you should think of using graphs

- Get Comfortable defining a graph class and writing BFS/DFS
  - Often times, graph problems themselves are just a BFS plus some trivial amount of extra work
  - The actual work comes from translating the data you are given into some representation of a graph
  - Most of time, you will be able to choose what type of implementation to use. I personally use Adjacency list or object collections whenever I can
  - Sometimes however, your data will come in as an adjaceny matrix, in that case you are best off using what you are given

- Graph problems are design problems!
  - As mentioned before, most graph problems are easy once you have a BFS/DFS and Graph representation written
  - You can generally add whatever extra infromation you want to the graph, consider using a HashMap<Vertex, T> to map extra attributes of type T to each vertex
  - Knowing how to convert a (Row, Column) index for a matrix to a single number index can be helpful. To do this, you take (row*number of columns per row) + Column

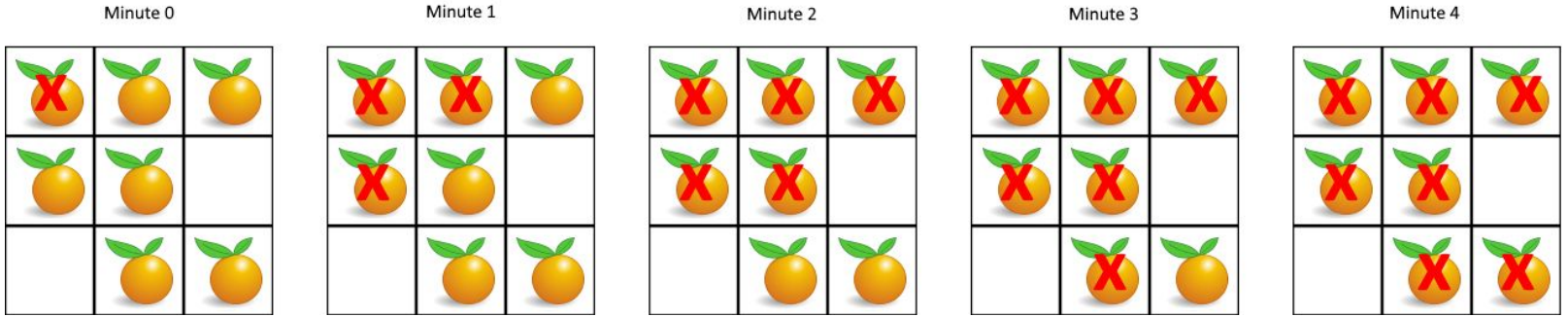
Sample Graph Problem

In a given grid, each cell can have one of three values:

- the value `0` representing an empty cell;
- the value `1` representing a fresh orange;
- the value `2` representing a rotten orange.

Every minute, any fresh orange that is adjacent (4-directionally) to a rotten orange becomes rotten.

Return the minimum number of minutes that must elapse until no cell has a fresh orange. If this is impossible, return `-1` instead.



Problem Approach

The first part of a graph problem is to decide how to represent the graph

Idea: an orange to its 4 neighboring squares, but only if that square has an orange in it

Once we have the graph, we need to think of how to solve the rest of the problem

Notice: all oranges adjacent to a rotten orange will be rotten minute 1, the ones 2 away will be rotten minute 2 etc.

This looks like a good candidate for breadth first search

1: Create a graph with all the oranges

2: Starting from each rotten orange, perform a breadth first search, keeping track of the distance of all fresh oranges. If the distance to this fresh orange is the smallest distance to a rotten orange so far, update the distance

3: If there are any fresh oranges left, then return -1, otherwise return the max distance of the oranges.

Pseudocode for This Problem

Create a graph representation, connecting all adjacent oranges

Initialize the distance of all rotten oranges to be 0, and the distance for all fresh oranges to be MAX_INT

For every rotten orange R

    Perform a breadth first search

    If you encount a fresh apple, and its current distance is greater than the distance to R, update the distance

Initialize max to be 0

For every Orange O

    If O is fresh return -1

    Else if O has the largest distance we have seen, store it in max

Return max

Java Code for this problem (Part 1)

Adjacency List Vertex Class Definition

```
private class Vertex {
    Vertex() {
        outgoing = new ArrayList<Vertex>();
        visited = false;
        time = Integer.MAX_VALUE;
    }
    public ArrayList<Vertex> outgoing;
    //some extra class variables used for specifics
    //of this problem
    public boolean visited;
    public int rotten;
    public int time;
}
```

Java Code for this problem (Part 2)

Graph Representation Initialization part 1
Create a vertex for every orange

```
HashMap<Integer,Vertex> graph = new HashMap<Integer,Vertex>();
Queue<Vertex> toVisit = new LinkedList<Vertex>();
//go through every index of the grid
for (int i=0; i<grid.length; i++) {
    for (int j=0; j<grid[i].length; j++) {
        //ignore the zero grids
        if (grid[i][j] == 0) {
            continue;
        }
        Vertex v = new Vertex();
        v.rotten = grid[i][j];
        //extra initialization for rotten oranges
        if (v.rotten == 2) {
            v.time = 0;
            toVisit.add(v);
        }
        //use conversion from (row,column) to greater row indecxing;
        //so I can have my hashtable have integer keys
        graph.put(i*grid[i].length+j,v);
```

Graph Representation Initialization part 1
Connect the vertex to its neighbors

```
graph.put(i*grid[i].length+j,v);
//if i>0, add the upper neighbor if it exists
if (i>0) {
    //if the key corresponding to index row i-1 column j has an orange
    if (graph.containsKey((i-1)*grid[i-1].length+j)) {
        //add an edge between both vertices
        graph.get((i-1)*grid[i-1].length+j).outgoing.add(v);
        v.outgoing.add(graph.get((i-1)*grid[i-1].length+j));
    }
}
//if j>o, add the left neighbor if it exists
if (j>0) {
    //if the key corresponding to index row i column j-1 has an orange
    if (graph.containsKey(i*grid[i].length+j-1)) {
        //add an edge between both vertices
        graph.get(i*grid[i].length+j-1).outgoing.add(v);
        v.outgoing.add(graph.get(i*grid[i].length+j-1));
    }
}
//NOTICE: If everyone adds their upper and left neighbor,
//then we never need to worry about lower and right neighbor
//because if you have a lower or right neighbor, they will add you
```

Java Code for this problem (Part 3)

```
//BFS
while(toVisit.peek() != null) {
    Vertex v = toVisit.poll();
    if (v.visited) {
        continue;
    }
    v.visited = true;
    for (Vertex v2: v.outgoing) {
        //if this is a shorter distance than what he have before
        //update it and all its neighbors
        if (v2.time > 1+v.time) {
            v2.time = 1+v.time;
            v2.visited = false;
            v2.rotten = 2;
            toVisit.add(v2);
        }
    }
}
int max = 0;
for (Vertex v: graph.values()) {
    //if not all the apples are rotten
    if (v.rotten == 1) {
        return -1;
    }
    if (v.time > max) {
        max = v.time;
    }
}
return max;
```

Breadth first search and solution

Graph Problems to Work On

Links to Problems 1-3 can be found on my Github account (Dane8373)

Direct Link: https://github.com/dane8373/SIG_Algorithm_Challenges/tree/master/Week7

1) There are N rooms and you start in room 0. Each room has a distinct number in 0, 1, 2, ..., N-1, and each room may have some keys to access the next room.Initially, all the rooms start locked (except for room 0).

You can walk back and forth between rooms freely. Return true if and only if you can enter every room.

2) if A is a direct friend of B, and B is a direct friend of C, then A is an indirect friend of C. And we define a friend circle is a group of students who are direct or indirect friends.

Given a N*N matrix M representing the friend relationship between students in the class. If M[i][j] = 1, then the i_{th} and j_{th} students are direct friends with each other, otherwise not. And you have to output the total number of friend circles among all the students.

3) You are given a list of cities. There are roads between these cities, but some of them were destroyed and need to be repaired. You are given a list of all the currently available roads, as well as the broken roads and the cost to repair them. Return the minimum cost that can be paid to make all the cities connected again

Problem Approach for Level 1 Problem

In this problem, all the doors can be accessed from any other (if you have the key) so no extra graph representation is needed

Use a stack to keep track of the rooms to visit

1: Put room 0 on the stack

2: while the stack is not empty, let the current node be the top of the stack

3: for every key to room i in the current room, add room i to the stack if we haven’t already visited it

4: Return true if all the rooms are visited after this

Java Code for level 1 problem (part 1)

```java
class Solution {
    public boolean canVisitAllRooms(List<List<Integer>> rooms) {
        //hashset to keep track of which rooms we visited
        HashSet<Integer> visited = new HashSet<Integer>();
        Stack<Integer> s = new Stack<Integer>();
        s.add(0);
        //DFS
        while (!s.empty()) {
            int curr = s.pop();
            visited.add(curr);
            for (int i: rooms.get(curr)) {
                if (!visited.contains(i)) {
                    s.add(i);
                }
            }
        }
        //if the visited hashset is the same size as the room list
        //then all rooms were visited
        return visited.size() == rooms.size();
    }
}
```

Problem Approach for Level 2 Problem

This problem is already given to us an adjacency matrix, it looks like a great candidate forDFS

Idea: at first there will be no friend groups, but all the people belong to one. Add all the people onto the stack, and do a DFS starting with the top

1: Put all row indexes on the stack

2: While the stack is not empty, make the current row equal to the top of the stack

3: if this row has already been visited, it has already been counted in some other group, and it doesn't need to be processes

4: Otherwise, increase the group count by 1

5: Go through all the columns of the current row, and add the column index to the stack if the value at that column is 1

6: return the group size

Java Code for level 2 problem (part 1)

```
class Solution {
    public int findCircleNum(int[][] M) {
        int groupCount = 0;
        HashSet<Integer> visited = new HashSet<Integer>();
        Stack<Integer> s = new Stack<Integer>();
        //add all rows to the stack
        for (int i=0; i<M.length; i++) {
            s.push(i);
        }
        //DFS
        while (!s.empty()) {
            int i = s.pop();
            //increase the group count if we haven't already put
            //this person in a group, make a new group for them
            if (!visited.contains(i)) {
                groupCount++;
            }
            visited.add(i);
            for (int j =0; j<M[i].length; j++) {
                if (M[i][j] == 1 && !visited.contains(j)) {
                    visited.add(j);
                    s.push(j);
                }
            }
        }
        return groupCount;
    }
}
```

Problem Approach for Level 3 Problem

For this problem, the information doesn't come in as a graph already, so you have to build one

Observation: This question is asking us for a minimum spanning tree. One useful fact about MSTs is that it doesn't matter what node you start on

Idea: Create a vertex for every city, add an edge of cost 0 for every road, and an edge with cost C for every road that needs to be repaired at cost C

Perform dijkstra's algorithm starting from whatever node you want and construct a minimum spanning tree of this newly build graph

1: Create the graph as described using an adjacency list. Initialize all costs of each vertex to infinity

2: Pick a random node, assign it a cost of 0 and add it to the priority queue

3: while the priority queue is not empty, make the current vertex be the front of the queue

4: For every vertex v adjacent to the current Vertex, if v's cost is greater than the current node's cost + the edge to v's cost, then update v's cost and add v to the queue. Return the sum of all the vertex costs

Java Code for level 3 problem (part 1)

Vertex and Edge classes

```
private static class Vertex {
    Vertex() {
        outgoing = new HashMap<Vertex,Edge>();
        visited = false;
        cost = Integer.MAX_VALUE;
        parent = null;
    }
    public HashMap<Vertex,Edge> outgoing;
    //used for djikstra algorithm;
    public boolean visited;
    public int cost;
    public Vertex parent;
}
private static class Edge {
    public int cost;
    //edges are bidirectional but I call it source/dest anyway
    public Vertex source;
    public Vertex dest;
}
```

Java Code for level 3 problem (part 2)

Graph initialization

```
public static int pathSolver(int numTotalAvailableCities, int numTotalAvailableRoads,
        ArrayList<ArrayList<Integer>> roadsAvailable, int numRoadsToBeRepaired,
        ArrayList<ArrayList<Integer>> costRoadsToBeRepaired) {
    ArrayList<Vertex> graph = new ArrayList<Vertex>();
    //create a vertex for every city
    for (int i = 0; i <= numTotalAvailableCities; i++) {
        graph.add(new Vertex());
    }
    //create and edge for every road
    for (int i=0; i<numTotalAvailableRoads; i++) {
        Edge e1 = new Edge();
        Edge e2 = new Edge();
        e1.cost = 0;
        //represent our bidirectional roads by adding
        //the edge to the source and the dest
        int source = roadsAvailable.get(i).get(0);
        int dest = roadsAvailable.get(i).get(1);
        e1.source = graph.get(source);
        e1.dest = graph.get(dest);
        e2.cost = 0;
        e2.source = graph.get(dest);
        e2.dest = graph.get(source);
        graph.get(source).outgoing.put(e1.dest,e1);
        graph.get(dest).outgoing.put(e2.dest,e2);
    }
```

Update the costs of the broken roads

```
//update the cost for all the broken roads
for (int i=0; i<numRoadsToBeRepaired; i++) {
    int source = costRoadsToBeRepaired.get(i).get(0);
    int dest = costRoadsToBeRepaired.get(i).get(1);
    int cost = costRoadsToBeRepaired.get(i).get(2);
    graph.get(source).outgoing.get(graph.get(dest)).cost=cost;
    graph.get(dest).outgoing.get(graph.get(source)).cost=cost;
}
```

Java Code for level 3 problem (part 3)

Djikstras Algorithm

```
//set up djikstras algorithm
PriorityQueue<Vertex> q =
        new PriorityQueue<Vertex>(16, Comparator.comparing((Vertex v) -> v.cost));
graph.get(1).cost = 0;
q.add(graph.get(1));
//djikstras algorithm
while(q.peek() != null) {
    Vertex v = q.poll();
    if (v.visited) {
        continue;
    }
    v.visited = true;
    for (Edge e: v.outgoing.values()) {
        if (!visitedVertexes.contains(e.dest) && v.cost + e.cost < e.dest.cost) {
            e.dest.cost = v.cost+e.cost;
            e.dest.parent = v;
            q.add(e.dest);
        }
    }
}
```

Extract the costs afterwards

```
int total = 0;
for (int i=1; i<graph.size(); i++) {
    //if it isn't connected (i.e., no road to one citiy)
    //return -1
    if (graph.get(i).visited == false) {
        return -1;
    }
    total+=graph.get(i).cost;
    //the cost of a vertex is the total path from the source
    //but this question just wants the total cost of the edges
    //so we subtract away the cost of the parent to account for this
    if (graph.get(i).parent!= null) {
        total-=graph.get(i).parent.cost;
    }
}
return total;
```

Next Week: Spring Break!