Lab walk-through #10 - 2

Topic **Advanced Graph Algorithms**

1. Introduction

This is a continuation of Lab 10 from last week on graphs and their algorithms. Here, the focus will be on directed graphs, minimum-spanning trees and sorting and shortest-path algorithms for graphs.

2. Lab Objectives

By the end of this lab the student will learn how to implement and study various graphing algorithms and their applications.

3. Lab Setup

Before beginning this lab, you should have:

1. The Eclipse and Java Runtime Environment on your computer
2. Completed the previous searching algorithms lab and its practice questions
3. Read Chapter 4 in your textbook

4. Lab Exercise

Directed graphs have edges that are one-way: the pair of vertices that defines each edge are **ordered**. Many applications are naturally expressed as in terms of directed graphs, such as internet connections, phone calls, paper citations in academia, and stock transactions.

The **outdegree** is the number of edges going from a vertex. The **indegree** of a vertex is the number of edges leading to the vertex. The first vertex in a directed edge is called its **head**, the second is its **tail** and the notation v->w is used to represent this relationship. A **directed path** in a digraph is a sequence of vertices in which there is a (directed) edge pointing from each vertex in the sequence to its successor in the sequence. We say that a vertex w is **reachable** from a vertex v if there is a directed path from v to w. Each vertex is reachable from itself.

Section 1: An API for a Digraph

What follows is the API for a directed graph and is almost identical to the undirected graph you came across in walkthrough 10.1.

public class	Digraph	
	Digraph(int V)	*create a V-vertex digraph with no edges*
	Digraph(In in)	*read a digraph from input stream in*
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(int v, int w)	*add edge v->w to this digraph*
Iterable<Integer>	adj(int v)	*vertices connected to v by edges pointing from v*
Digraph	reverse()	*reverse of this digraph*
String	toString()	*string representation*

API for a digraph

As you can see, we still use the adjacency list representation we used for undirected graphs, but it is simpler since each edge appears just once. The linked list for each vertex contains all the nodes that can be reached from the vertex with one edge.

Task 1: Implement Digraph.java

Implement the API shown above. Create a Java project called "cas2xb3_lab102" and create a class called Digraph.java. Remember to use the Bag.java ADT provided to you on the textbook website. Add the following code to your Digraph.java:

```
public class Digraph{
	private final int V;
	private int E;
	private Bag<Integer>[] adj;

	public Digraph(int V){
		this.V = V;
		this.E = 0;
		adj = (Bag<Integer>[]) new Bag[V];

		for (int v = 0; v < V; v++)
			adj[v] = new Bag<Integer>();
	}

	public int V() { return V; }
	public int E() { return E; }

	public void addEdge(int v, int w){
		adj[v].add(w);
		E++;
	}

	public Iterable<Integer> adj(int v) { return adj[v]; }

	public Digraph reverse(){
		Digraph R = new Digraph(V);
		for (int v = 0; v < V; v++)
		for (int w : adj(v))
		R.addEdge(w, v);
		return R;
	}
}
```

Implement the other functions yourself, or add them in from the previous walkthrough.
As in our previous walkthrough, this is our base implementation, and all other algorithms or applications we develop should be put into separate classes.

Recall the reachability problem from the previous walkthrough. For digraphs, we ask the following question:

***Single-source reachability**: Is there a directed path from s to a given target vertex v?*

The solution to this is exactly the same as the reachability problem we tackled in the previous walkthrough and the API we will use to solve this is shown below.

public class DirectedDFS	
DirectedDFS(Digraph G, int s)	*find vertices in G that are reachable from s*
DirectedDFS(Digraph G, Iterable<Integer> sources)	*find vertices in G that are reachable from sources*
boolean marked(int v)	*is v reachable?*

API for reachability in digraphs

Task 2: Reachability in a directed graph

Implement the above API to carry out DFS on a digraph. Create a new class called DirectedDFS and add the following code to it (as shown on page 571 of your textbook). You can use their main method to test this implementation. Convince yourself that this works as intended.

```
public class DirectedDFS{
      private boolean[] marked;

      public DirectedDFS(Digraph G, int s){
            marked = new boolean[G.V()];
            dfs(G, s);
      }

      public DirectedDFS(Digraph G, Iterable<Integer> sources){
            marked = new boolean[G.V()];
            for (int s : sources)
            if (!marked[s]) dfs(G, s);
      }

      private void dfs(Digraph G, int v){
            marked[v] = true;
            for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
      }
      public boolean marked(int v){ return marked[v]; }
}
```

By adding a second constructor that takes a list of vertices, this API supports for clients the following generalization of the problem, that is, the multiple-source reachability problem. This problem arises in the classic string processing algorithm (section 5.4 in your textbook). This isn't exactly DFS as it just marks the vertices it visits, but this can be substituted for printing if needed.

Take a good look at the trace on page 572 to make sure you understand how DFS works on a digraph.

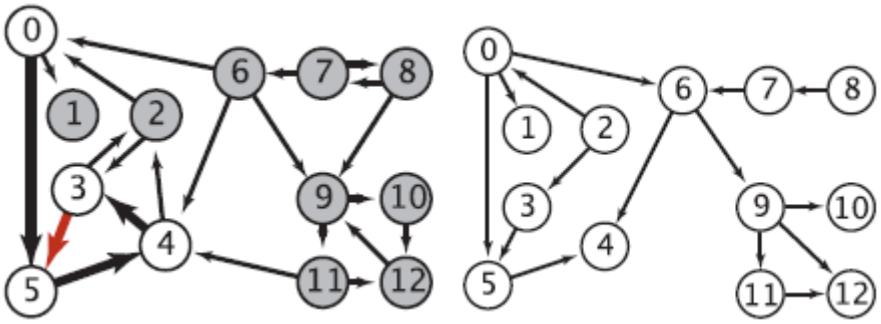
Directed cycles are of particular importance in applications that involve processing digraphs. Identifying directed cycles in a typical digraph can be a challenge. The simplest example is scheduling tasks, which is easily modelled as a digraph.

If job x must be completed before job y, job y before job z, and job z before job x, then someone has made a mistake, because those three constraints cannot all be satisfied. In general, if a precedence-constrained scheduling problem has a directed cycle, then there is no feasible solution. To check for such errors, we need to be able to solve the following problem:

Does a given digraph have a directed cycle? If so, find the vertices on some such cycle, in order from some vertex back to itself.

A **directed acyclic graph** (DAG) is a digraph with no directed cycles. We ask the question that is a particular digraph a DAG?

A directed graph *with* cycles is shown below on the left and *without* cycles is shown on the right.



How do we go about answering this question? We develop a depth-first-search-based solution to this problem. The recursive call stack maintained by the system represents the "current" directed path under consideration. If we ever find a directed edge v->w to a vertex w that is on that stack, we have found a cycle, since the stack is evidence of a directed path from w to v, and the edge v->w completes the cycle. Moreover, the absence of any such *back edges* implies that the graph is acyclic.

Task 3: Check for cycles in a digraph

Create a new class called DirectedCycle and add the following code as shown below. Don't forget to import the Stack library.

```
public class DirectedCycle{
      private boolean[] marked;
      private int[] edgeTo;
      private Stack<Integer> cycle;//vertices on a cycle (if it exists)
      private boolean[] onStack; // vertices on recursive call stack

      public DirectedCycle(Digraph G){
            onStack = new boolean[G.V()];
            edgeTo = new int[G.V()];
```

```
            marked = new boolean[G.V()];
            for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G, v);
      }

      private void dfs(Digraph G, int v){…} ;//explained below

      public boolean hasCycle(){ return cycle != null; }
      public Iterable<Integer> cycle(){ return cycle; }
}
```

This class adds to our standard recursive dfs() a boolean array onStack[] to keep track of the vertices for which the recursive call has not completed. When it finds an edge v->w to a vertex w that is on the stack, it has discovered a directed cycle, which it can recover by following edgeTo[] links.

When executing dfs(G, v), we have followed a directed path from the source to v. To keep track of this path, DirectedCycle maintains a vertex-indexed array onStack[] that marks the vertices on the recursive call stack (by setting onStack[v] to true on entry to dfs(G, v) and to false on exit).

DirectedCycle also maintains an edgeTo[] array so that it can return the cycle when it is detected, in the same way as DepthFirstPaths (page 536) and BreadthFirstPaths (page 540) return paths.

```
private void dfs(Digraph G, int v){
      onStack[v] = true;
      marked[v] = true;

      for (int w : G.adj(v))
            if (this.hasCycle()) return;
            else if (!marked[w]){ edgeTo[w] = v; dfs(G, w); }
            else if (onStack[w]){
                  cycle = new Stack<Integer>();
                  for (int x = v; x != w; x = edgeTo[x])
                  cycle.push(x);
                  cycle.push(w);
                  cycle.push(v);
            }
      onStack[v] = false;
}
```

Note that this algorithm returns the first cycle it finds rather than all the cycles. In principle there could be an exponential number of them, so we look for only one.

As in the previous walkthrough, the importance of graphs comes in traversing the vertices in a systematic order. The next step is to actually do an appropriate DFS in a particular order, and depending on the application you're interested in, there are three ways you can do this:

1. **Preorder** : Put the vertex on a queue before the recursive calls.
2. **Postorder** : Put the vertex on a queue after the recursive calls.
3. **Reverse postorder** : Put the vertex on a stack after the recursive calls.

Task 4: Implement Depth First Orderings on a given graph

Let us create a new class for this purpose. Call it DepthFirstOrder. It will contain two queues and a stack to carry out these three traversals.

```
private boolean[] marked;
private Queue<Integer> pre; // vertices in preorder
private Queue<Integer> post; // vertices in postorder
private Stack<Integer> reversePost; // vertices in reverse postorder
```

Instantiate these variables in the constructor and call dfs() if there's a vertex that hasn't been visited (marked). Then the implementation of dfs is trivial.

```
public DepthFirstOrder(Diagraph G){
      post = new Queue<Integer>();
      pre = new Queue<Integer>();
      marked = new boolean[G.V()];
      for(int v = 0; v<G.V(); v++){
            if(!marked[v])
                  dfs(G,v);
      }
}


private void dfs(Digraph G, int v){
      pre.enqueue(v);
      marked[v] = true;
      for (int w : G.adj(v)){
            if (!marked[w]){
                  dfs(G, w);
            }
      }
      post.enqueue(v);
      reversePost.push(v);
}
```

Convince yourself that this is correct by printing out the results. Do this by creating public helper methods as shown below.

```
public Iterable<Integer> pre(){ return pre; }
public Iterable<Integer> post(){ return post; }
```

```
public Iterable<Integer> reversePost(){ return reversePost; }
```

Task 5: Topological sort

What is meant by Topological sorting?

> *Given a digraph, put the vertices in order such that all its directed edges point from a vertex earlier in the order to a vertex later in the order (or report that doing so is not possible).*

A digraph has a topological order if and only if it is a DAG.

Therefore, in order to do a topological sort, first check if a digraph is a DAG. This is where our previous implementation comes in handy. Go ahead and implement topological sort in a new class called Topological.java and check your answers with the implementation on page 581 in your textbook.

Section 2 – Minimum Spanning Tree

This walkthrough will concentrate on the application of the MST algorithm, and then principles of Minimum Spanning trees will be left as an exercise for you to read up on, starting with page 604 in your textbook. A short introduction is included here for the whole picture.

A **spanning tree** of a graph is a connected sub-graph with no cycles that includes all the vertices. A **minimum spanning tree (MST)** of an edge-weighted graph is a spanning tree whose weight (the sum of the weights of its edges) is no larger than the weight of any other spanning tree. In this section, we examine two classical algorithms for computing MSTs: **Prim's algorithm** and **Kruskal's algorithm.**

MST algorithms are now important in the design of many types of networks (communication, electrical, hydraulic, computer, road, rail, air, and many others) and also in the study of biological, chemical, and physical networks that are found in nature.

There are a few assumptions we make:

1. The graph is connected.
2. The edge weights are not necessarily distances.
3. The edge weights may be 0 or negative.
4. The edge weights are all different.

In summary, we assume throughout the presentation that our job is to find the MST of a connected edge-weighted graph with arbitrary (but distinct) weights.

A **cut** of a graph is a partition of its vertices into two non-empty disjoint sets. A crossing edge of a cut is an edge that connects a vertex in one set with a vertex in the other. Typically, we specify a cut by specifying a set of vertices, leaving implicit the assumption that the cut comprises the given vertex set and its complement, so that a crossing edge is an edge from a vertex in the set to a vertex not in the set.

Now we will provide a principle that allows Prim's and Kruskal's algorithm to work. Read in your textbook for why this works:

> ***Given any cut in an edge-weighted graph, the crossing edge of minimum weight is in the MST of the graph.***

We make one final observation before moving onto the implementation: MST algorithms are always greedy. If you don't know what this means, search for the **greedy algorithm paradigm** in any algorithms textbook.

In order to implement the algorithms, we will need an **edge-weighted graph**. We accomplish this by creating an ADT called Edge and adding onto our Graph implementation right in the beginning of walkthrough 8.

The API for an Edge ADT looks like this:

public class Edge implements Comparable<Edge>		
	Edge(int v, int w, double weight)	*initializing constructor*
double	weight()	*weight of this edge*
int	either()	*either of this edge's vertices*
int	other(int v)	*the other vertex*
int	compareTo(Edge that)	*compare this edge to* e
String	toString()	*string representation*

API for a weighted edge

Task 6: Implement the Edge and EdgeWeightedGraph API
Both can be found on page 608 of your textbook and you can check your implementation with that on page 610.

Our choice to use explicit Edge objects leads to clear and compact client code(verify this on page 613). It carries a small price: each adjacency-list node has a reference to an Edge object, with redundant information (all the nodes on v's adjacency list have a v). We also pay object overhead cost. Although we have only one copy of each Edge, we do have two references to each Edge object. An alternative and widely used approach is to keep two list nodes corresponding to each edge, just as in Graph, each with a vertex and the edge weight in each list node. This alternative also carries a price—two nodes, including two copies of the weight for each edge.

Task 7: Prim's Algorithm - Lazy version
Our first MST method, known as Prim's algorithm, is to attach a new edge to a single growing tree at each step. Start with any vertex as a single-vertex tree; then add V-1 edges to it, always taking next (colouring black) the minimum weight edge that connects a vertex on the tree to a vertex not yet on the tree (a crossing edge for the cut defined by tree vertices). Remember to import the appropriate libraries when copying the code that follows.

We use three data structures to represent the graph:

1. **Vertices on the tree:** We use a vertex-indexed boolean array marked[], where marked[v] is true if v is on the tree.
2. **Edges on the tree:** We use one of two data structures: a queue mst to collect MST edges or a vertex-indexed array edgeTo[] of Edge objects, where edgeTo[v] is the Edge that connects v to the tree.
3. **Crossing edges:** We use a MinPQ<Edge> priority queue that compares edges by weight.

Implementation-wise, they can be represented using the following code:

```
private boolean[] marked; // MST vertices
private Queue<Edge> mst; // MST edges
private MinPQ<Edge> pq; // crossing (and ineligible) edges
```

Next we have to **maintain the set of crossing edges**. Each time that we add an edge to the tree, we also add a vertex to the tree. To maintain the set of crossing edges, we need to add to the priority queue all edges from that vertex to any non-tree vertex (using marked[] to identify such edges). But we must do more: any edge connecting the vertex just added to a tree vertex that is already on the priority queue now becomes ineligible (it is no longer a crossing edge because it connects two tree vertices).

An eager implementation of Prim's algorithm would remove such edges from the priority queue; we first consider a simpler lazy implementation of the algorithm where we leave such edges on the priority queue, deferring the eligibility test to when we remove them.

However, the question still remains: how do we (efficiently) find the crossing edge of minimal weight?

There are a number of ways of doing this, and we will start with a simplified one first. Create a new class called LazyPrimMST and implement Prim's algorithm in the constructor of the class so that the properties of the MST can be explored using other functions.

```
public LazyPrimMST(EdgeWeightedGraph G){
      pq = new MinPQ<Edge>();
      marked = new boolean[G.V()];
      mst = new Queue<Edge>();
      visit(G, 0); // assumes G is connected

      while (!pq.isEmpty()){
            Edge e = pq.delMin(); // Get lowest-weight
            int v = e.either(), w = e.other(v); // edge from pq.

            if (marked[v] && marked[w]) continue; // Skip if ineligible.
            mst.enqueue(e); // Add edge to tree.

            if (!marked[v]) visit(G, v); // Add vertex to tree
            if (!marked[w]) visit(G, w); // (either v or w).
      }
}
```

The inner loop takes an edge from the priority queue and (if it is not ineligible) add it to the tree, and also add to the tree the new vertex that it leads to, updating the set of crossing edges by calling visit() with that vertex as argument. The weight() method requires iterating through the tree edges to add up the edge weights (lazy approach) or keeping a running total in an instance variable (eager approach).

We use a private method visit() that puts a vertex on the tree, by marking it as visited and then putting all of its incident edges that are not ineligible onto the priority queue, thus ensuring that the priority

queue contains the crossing edges from tree vertices to non-tree vertices (perhaps also some ineligible edges).

```
private void visit(EdgeWeightedGraph G, int v){
      // Mark v and add to pq all edges from v to unmarked vertices.
      marked[v] = true;
      for (Edge e : G.adj(v))
            if (!marked[e.other(v)]) pq.insert(e);
}
```

Task 8: Prim's Algorithm - Eager version

To improve the LazyPrimMST(call the new class PrimMST.java), we might try to delete ineligible edges from the priority queue, so that the priority queue contains only the crossing edges between tree vertices and non-tree vertices. But we can eliminate even more edges.

The key is to note that our only interest is in the **minimal edge** from each non-tree vertex to a tree vertex. When we add a vertex v to the tree, the only possible change with respect to each non-tree vertex w is that adding v brings w closer than before to the tree. In short, we do not need to keep on the priority queue all of the edges from w to tree vertices—we just need to keep track of the minimum-weight edge and check whether the addition of v to the tree necessitates that we update that minimum (because of an edge v-w that has lower weight), which we can do as we process each edge in v's adjacency list.

In other words, we maintain on the priority queue just one edge for each non-tree vertex w: the shortest edge that connects it to the tree. Any longer edge connecting w to the tree will become ineligible at some point, so there is no need to keep it on the priority queue.

We now replace our previous three variables with the following:

```
private Edge[] edgeTo; // shortest edge from tree vertex
private double[] distTo; // distTo[w] = edgeTo[w].weight()
private boolean[] marked; // true if v on tree
private IndexMinPQ<Double> pq; // eligible crossing edges
```

So if v is not on the tree but has at least one edge connecting it to the tree, then edgeTo[v] is the shortest edge connecting v to the tree, and distTo[v] is the weight of that edge. Also, all such vertices v are maintained on the index priority queue, as an index v associated with the weight of edgeTo[v]. This implies that **the minimum key on the priority queue is the weight of the minimal-weight crossing edge, and its associated vertex v is the next to add to the tree.**

Now add the following code to your class PrimMST.java:

```
public PrimMST(EdgeWeightedGraph G){
      edgeTo = new Edge[G.V()];
      distTo = new double[G.V()];
      marked = new boolean[G.V()];
```

```
      for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;

      pq = new IndexMinPQ<Double>(G.V());
      distTo[0] = 0.0;
      pq.insert(0, 0.0); // Initialize pq with 0, weight 0.

      while (!pq.isEmpty())
            visit(G, pq.delMin()); // Add closest vertex to tree.
}
      private void visit(EdgeWeightedGraph G, int v){
      // Add v to tree; update data structures.
      marked[v] = true;
      for (Edge e : G.adj(v)){
            int w = e.other(v);
            if (marked[w]) continue; // v-w is ineligible.
                  if (e.weight() < distTo[w]){
                        // Edge e is new best connection from tree to w.
                        edgeTo[w] = e;
                        distTo[w] = e.weight();

                        if (pq.contains(w)) pq.change(w, distTo[w]);
                        else pq.insert(w, distTo[w]);
                  }
      }
}
```

There are two methods missing here – edges() and weight(). Implement them as an exercise. Can you test this algorithm with client code? Do this now.

Task 9: Kruskal's Algorithm

Prim's algorithm builds the MST one edge at a time, finding a new edge to attach to a single growing tree at each step. Kruskal's algorithm also builds the MST one edge at a time; but, by contrast, it finds an edge that connects two trees in a forest of growing trees. We start with a degenerate forest of V single-vertex trees and perform the operation of combining two trees (using the shortest edge possible) until there is just one tree left: the MST.

To implement this algorithm, we use a priority queue to consider the edges in order by weight, a union-find data structure to identify those that cause cycles, and a queue to collect the MST edges. The weight() method requires iterating through the queue to add the edge weights. Implement this in your class called KruskalMST.java, as shown below. Remember to import the UnionFind and MinPQ libraries found on the textbook website.

```
public class KruskalMST{
      private Queue<Edge> mst;
```

```
    public KruskalMST(EdgeWeightedGraph G){
        mst = new Queue<Edge>();
        MinPQ<Edge> pq = new MinPQ<Edge>(G.edges());
        UF uf = new UF(G.V());

        while (!pq.isEmpty() && mst.size() < G.V()-1){
            Edge e = pq.delMin(); // Get min weight edge on pq
            int v = e.either(), w = e.other(v); //and vertices.
            if (uf.connected(v, w))
                continue; //Ignore ineligible edges.
            uf.union(v, w); // Merge components.
            mst.enqueue(e); // Add edge to mst.
        }
    }
    public Iterable<Edge> edges(){ return mst; }
    public double weight() // Complete this.
}
```

This implementation of Kruskal's algorithm uses a queue to hold MST edges, a priority queue to hold edges not yet examined, and a union-find data structure for identifying ineligible edges. The MST edges are returned to the client in increasing order of their weights.

Think about the order of growth of these algorithms you have implemented. Which one is "better"? Well, it depends on the structure of the graph.

For most graphs, the cost of finding the MST is only slightly higher than the cost of extracting the graph's edges. This rule holds except for huge graphs that are extremely sparse, but the available performance improvement over the best-known algorithms even in this case is a small constant factor, perhaps a factor of 10 at best.

Section 3 – Shortest Path

All the algorithms we have studied so far are small sub-sections of problems in the real world. Or algorithms that give us more insight into how a graph is structured. However, the largest use of graphs in almost any application is to answer the following question:

Find the lowest-cost way to get from one vertex to another.

A **shortest path** from vertex s to vertex t in an edge-weighted digraph is a directed path from s to t with the property that no other such path has a lower weight.

The data structures that we need to represent shortest paths are straightforward:

1. **Edges on the shortest-paths tree**: As for DFS, BFS, and Prim's algorithm, we use a parent-edge representation in the form of a vertex-indexed array edgeTo[] of DirectedEdge objects, where edgeTo[v] is edge that connects v to its parent in the tree (the last edge on a shortest path from s to v).
2. **Distance to the source**: We use a vertex-indexed array distTo[] such that distTo[v] is the length of the shortest known path from s to v.

Task 10: Dijkstra's Algorithm implementation

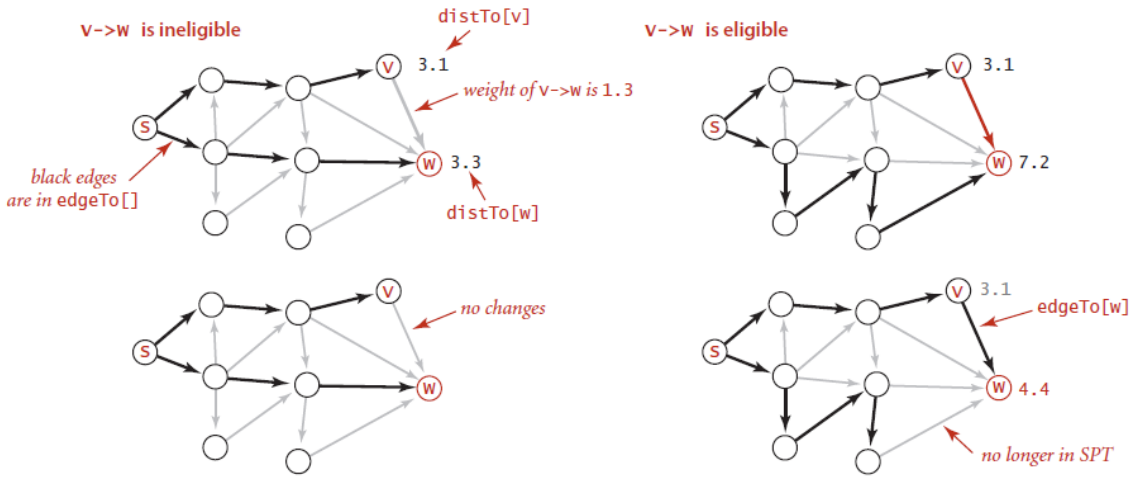
Create a class called DijkstraSP and start with the following instance variables:

```
private DirectedEdge[] edgeTo;
private double[] distTo;
private IndexMinPQ<Double> pq;
```

By convention, edgeTo[s] is `null` and distTo[s] is `0`. We also adopt the convention that distances to vertices that are not reachable from the source are all `Double.POSITIVE_INFINITY`.

Our shortest-paths implementations are based on a simple operation known as **relaxation**. To relax an edge v->w means to test whether the best known way from s to w is to go from s to v, then take the edge from v to w, and, if so, update our data structures to indicate that to be the case.

The best known distance to w through v is the **sum** of distTo[v] and e.weight()— if that value is not smaller than distTo[w], we say the edge is **ineligible**, and we ignore it; if it is smaller, we update the data structures.



Edge relaxation (two cases)

However, this is for a particular edge. We need to iterate through all of the vertices and make sure we relax all edges pointing from a given vertex, as shown below. Add this method to your DijkstraSP.java. Remember to add on the `IndexMinPQ` library from the textbook website.

```
private void relax(EdgeWeightedDigraph G, int v){
    for(DirectedEdge e : G.adj(v)){
        int w = e.to();
        if (distTo[w] > distTo[v] + e.weight()){
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
            if (pq.contains(w)) pq.changeKey(w, distTo[w]);
            else pq.insert(w, distTo[w]);
        }
```

```
        }
}
```

Note that any edge from a vertex whose distTo[v] entry is finite to a vertex whose distTo[] entry is infinite is eligible and will be added to edgeTo[] if relaxed. In particular, some edge leaving the source is the first to be added to edgeTo[].

Add a few more client code methods, as found on page 649. These allow you to make the best use of this class once implemented. Now we are ready to define the constructor of DijkstraSP.

```
public DijkstraSP(EdgeWeightedDigraph G, int s)
{
      edgeTo = new DirectedEdge[G.V()];
      distTo = new double[G.V()];
      pq = new IndexMinPQ<Double>(G.V());

      for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
      distTo[s] = 0.0;
      pq.insert(s, 0.0);

      while (!pq.isEmpty())
            relax(G, pq.delMin());
}
```

This implementation of Dijkstra's algorithm grows the SPT by adding an edge at a time, always choosing the edge from a tree vertex to a non-tree vertex whose destination w is closest to s. Does this make sense? What is the order of growth of this algorithm?

This can now help us solve the more generalised problem called **all-pairs shortest paths**:

Given an edge-weighted digraph, support queries of the form Given a source vertex s and a target vertex t, is there a path from s to t? If so, find a shortest such path (one whose total weight is minimal).

Can you work out how to do this? You just have to iterate over all the vertices in the graph and call the DijkstraSP constructor each time.

Can you work out how to solve the SP problem even faster? By relaxing vertices in topological order, we can solve the single source shortest-paths problem for edge-weighted DAGs in time proportional to $E + V$.

6. Further Practice Problems

For further practice, here are some suggested questions from your textbook:
4.2.7, 4.2.9,4.2.20, 4.3.22, 4.3.41