Names: Ahmed Cheema, Havin Lim
Assignment name: Lab 36
Assignment due date: Sunday
Written/online sources used:
Help obtained: none
"We confirm that the above list of sources is complete AND that We have not talked to anyone else about the solution to this problem."

Preparation

1. The dump method uses vertex numbers, which will not be easily understandable to the user.
2. In this graph, edges are stored and added given two or three inputs (from and to, possibly weight). Vertices are also stored and added and given one input (the vertex).
3. If working with directed graphs, we will need to ensure that we are following the direction of the edges. This means that all traversions will have to go from the "from" vertex of an edge to the "to" vertex of an edge, and not vice versa.
4. If working with undirected graphs, we will need to consider both possible traversions of an edge - "from" to "to" or "to" to "from." And with the graph being weighted as well, the cost of a traversion will need to be accounted for in order to ensure that the most efficient paths are chosen. This will require the total cost to be stored.

readGraph

```
/**
 * Read graph from file
 * @param fName
 */
public void readGraph(String fName) {
	try {
		Scanner sc = new Scanner(new File(fName));
		while (sc.hasNextLine()) {
			String line = sc.nextLine();
			char[] values = line.toCharArray();
			if (values.length > 0) {
				String v1 = Character.toString(values[0]);
				String v2 = Character.toString(values[2]);
				int wt = Character.getNumericValue(values[4]);

				if (this.vertexNumber(v1) == -1) {
					this.addVertex(v1);
				}
				if (this.vertexNumber(v2) == -1) {
					this.addVertex(v2);
				}

				this.addEdge(v1,v2,wt);
			}
			else {
```

```
                        continue;
                    }
                }
                sc.close();
            }
            catch (FileNotFoundException e) {
              e.printStackTrace();
            }
            catch (Exception e) {
              e.printStackTrace();
            }
      }
```

reachableFrom

```
      /**
       * Prints out a list of all the vertices reachable from a starting point
       */
      public void reachableFrom(PrintWriter pen, int vertex) {
            ArrayList<Integer> processed = new ArrayList<Integer>();

            this.reachableFromHelper(pen,vertex,processed);
      }

      /**
       * Prints out a list of all the vertices reachable from a starting point
       */
      public void reachableFromHelper(PrintWriter pen, int vertex, ArrayList<Integer>
processed) {
            Iterator<Edge> iter = this.edgesFrom(vertex);

            while (iter.hasNext()) {
                  Edge e = iter.next();
                  if (processed.contains(e.to())) {
                        continue;
                  }
                  else {
                        processed.add(e.to());
                        System.out.println(vertexName(e.to()));
                        reachableFromHelper(pen,e.to(),processed);
                  }
            }
      }
```

pathsFrom

```
      /**
       * Prints out a path to each vertex reachable from the input vertex
       * @param pen PrintWriter object
       * @param vertex Index for the start vertex
       */
      public void pathsFrom(PrintWriter pen, int vertex) {
            // Iterate through all vertices
            for (int end = 0; end < this.numVertices; end++) {
```

```
                // Get path from vertex to destination vertex
                String path = findPath(vertex, end);

                // If it's not null and greater than 1, print
                if (path != null && path.length() > 1) {
                      pen.println(path);
                }
        }
  }

  /**
   * Return path between start and destination indices
   */
  public String findPath(int start, int end) {
          // ArrayList of processed integers
          ArrayList<Integer> processed = new ArrayList<Integer>();

          // String array of the local path
          String[] localPath = new String[this.numVertices()];

          // add parent vertex to processed
          processed.add(start);

          // add parent vertex to localPath
          localPath[start] = this.vertexName(start);

          // while processed not empty, iterate
          while(!processed.isEmpty()) {
             // remove last element from processed and feed it back into helper until
destination reached
               if (findPathHelper(processed.remove(processed.size()-1), end,
processed, localPath)) {
                     break;
               }
          }
          return localPath[end];
       }

  /**
   * Recursive function that iterates through graph seeking a match between start and
end vertices
   * Helper for findPath method
   */
  public boolean findPathHelper(int start, int end, ArrayList<Integer> processed,
String[] localPath) {
              // start and end converge - path traversal complete
              if (start == end) {
                return true;
          }

        // get iterator of edges from parent index
        Iterator<Edge> iter = this.edgesFrom(start);

        // iterate through edges from parent index
        while(iter.hasNext()) {
```

```
             // get index of vertex on edge
            int nextVertex = iter.next().to();

            // add string to localPath
            localPath[nextVertex] = localPath[start] + " -> " +
this.vertexName(nextVertex);

            // add next vertex to processed
            processed.add(nextVertex);
        }
        // no match
           return false;
    }
```

Tester

```
  public static void main(String[] args) throws Exception {
        PrintWriter pen = new PrintWriter(System.out, true);
        Graph g = new Graph("Graph.txt");

        System.out.println("Printing vertices reachable from vertex 0:");
        g.reachableFrom(pen, 0);

        System.out.println("\nPrinting all paths from vertex 0:");

        g.pathsFrom(pen, 0);
  }
```

Output

```
Printing vertices reachable from vertex 0:
B
E
C
F
D

Printing all paths from vertex 0:
A -> B
A -> D
A -> D -> E
A -> D -> E -> C
A -> D -> E -> F
```

Names: Ahmed Cheema, Havin Lim
Assignment name: Lab 37
Assignment due date: Sunday
Written/online sources used: none
Help obtained: none
"We confirm that the above list of sources is complete AND that We have not talked to anyone else about the solution to this problem."

Preparation

1. We will need to implement the printPath methods. Unreachables are currently not handled and will need to be taken care of.
2. Edges are a class that have a Vertex field for destination and a double field for cost. Furthermore, the Vertex class has a field (a List of Edge objects) for adjacent vertices corresponding to that Vertex object.
3. The implementation is different than the last lab because the Edge class no longer has separate fields for the "from" and "to" vertices. Rather, there is simply a "dest" (destination) field for a Vertex. Also, the Vertex class has a field representing a List of Edge objects adjacent to that Vertex object. This was not the case in the previous lab's implementation, which did not even have a Vertex class.

```
// printPath public method
public void printPath(String destName) throws NoSuchElementException
	{
		Vertex v = vertexMap.get(destName);
		if(v == null) {
			System.out.println("The input vertex does not exist.");
			throw new NoSuchElementException();
		}
		else {
			printPath(v);
			System.out.println("Cost: " + v.dist);
		}
	}



// printPath private method
private void printPath( Vertex dest ) {

		if (dest.prev == null) {
			System.out.println(dest.name);

		}

		else {
```

```
                printPath(dest.prev);
                System.out.println(dest.name);

        }
    }




// dijkstra method
    public void dijkstra( String startName )
    {
        PriorityQueue<Path> pq = new PriorityQueue<Path>( );

        Vertex start = vertexMap.get( startName );
        if( start == null )
            throw new NoSuchElementException( "Start vertex not found" );

        clearAll( );
        pq.add( new Path( start, 0 ) );
        start.dist = 0;

        int nodesSeen = 0;
        while( !pq.isEmpty( ) && nodesSeen < vertexMap.size( ) )
        {
            Path vrec = pq.remove( );
            Vertex v = vrec.dest;
            if( v.scratch != 0 )  // already processed v
                continue;

            v.scratch = 1;
            nodesSeen++;

            for( Edge e : v.adj )
            {
                Vertex w = e.dest;
                double cvw = e.cost;

                if( cvw < 0 )
                    throw new GraphException( "Graph has negative
edges" );

                if( w.dist > v.dist + cvw )
                {
                    w.dist = v.dist + cvw;
                    w.prev = v;
                    pq.add( new Path( w, w.dist ) );
                }
```

```
                        else if(w.dist == v.dist + cvw) {
                              int oldLength = 0;
                              int newLength = 1;

                              Vertex a;
                              a = w;
                              while(a != null) {
                                    a = a.prev;
                                    oldLength += 1;
                              }

                              a = v;
                              while(a != null) {
                                    a = a.prev;
                                    newLength += 1;
                              }

                              if(newLength < oldLength) {
                                    w.dist = v.dist + cvw;
                                    w.prev = v;
                                    pq.add( new Path( w, w.dist ) );
                              }

                        }
                  }
            }
      }

// Test results
// Graph.txt file read
File read...
5 vertices
Enter start node:A
Enter destination node:D
Enter algorithm (u, d, n, a, c): d
A
B
D
Cost: 12.0
A
C
E
D
Cost: 12.0
```

Names: Ahmed Cheema, Havin Lim
Assignment name: Lab 38
Assignment due date: Sunday
Written/online sources used: none
Help obtained: none
"We confirm that the above list of sources is complete AND that We have not talked to anyone else about the solution to this problem."

```
// criticalPath method (same as longestPath)

public void criticalPath(String startName)
      {
            Vertex start = vertexMap.get( startName );
            if( start == null )
                  throw new NoSuchElementException( "Start vertex not found" );

            clearAll( );
            Queue<Vertex> q = new LinkedList<Vertex>( );
            start.dist = 0;

            // Compute the indegrees
            Collection<Vertex> vertexSet = vertexMap.values( );
            for( Vertex v : vertexSet )
                  for( Edge e : v.adj )
                        e.dest.scratch++;

            // Enqueue vertices of indegree zero
            for( Vertex v : vertexSet )
                  if( v.scratch == 0 )
                        q.add( v );

            int iterations;
            for( iterations = 0; !q.isEmpty( ); iterations++ )
            {
                  Vertex v = q.remove( );

                  for( Edge e : v.adj )
                  {
                        Vertex w = e.dest;
                        double cvw = e.cost;

                        if( --w.scratch == 0 )
                              q.add( w );

                        if( v.dist == INFINITY )
                              continue;
```

```
                    if(w.dist == INFINITY || w.dist < v.dist + cvw)

                    {
                          w.dist = v.dist + cvw;
                          w.prev = v;
                    }
              }
        }

        if( iterations != vertexMap.size( ) )
              throw new GraphException( "Graph has a cycle!" );
}

/**
 * Process a request; return false if end of file.
 */
public static boolean processRequest( Scanner in, Graph g )
{
        try
        {
              System.out.print( "Enter start node:" );
              String startName = in.nextLine( );

              System.out.print( "Enter destination node:" );
              String destName = in.nextLine( );

              System.out.print( "Enter algorithm (u, d, n, a, c): " );
              String alg = in.nextLine( );

              if( alg.equals( "u" ) )
                    g.unweighted( startName );
              else if( alg.equals( "d" ) )
              {
                    g.dijkstra( startName );
                    g.printPath( destName );
                    g.dijkstra2( startName );
              }
              else if( alg.equals( "n" ) )
                    g.negative( startName );
              else if( alg.equals( "a" ) )
                    g.acyclic( startName );
              else if( alg.equals( "c" ) )
                    g.criticalPath(startName);

              g.printPath( destName );
        }
        catch( NoSuchElementException e )
        { return false; }
```

```
            catch( GraphException e )
            { System.err.println( e ); }
            return true;
      }

// Test results
// Graph2.txt file read
File read...
13 vertices
Enter start node:A
Enter destination node:I
Enter algorithm (u, d, n, a, c): c
A
B
C
D
E
G
I
Cost: 19.0
```