Princeton University
Department of Operations Research
and Financial Engineering

ORF 201
Computer Methods in Problem Solving

Lab 9: Shortest Paths in a Network

We will not be doing this lab this year!!!

---

1. Introduction

The objective of this assignment is to implement an algorithm to solve *shortest-path-tree* problems on networks.

A *network* (or graph) $\mathcal{G} = \{ \mathcal{N}, \mathcal{A} \}$ consists of a set of nodes $\mathcal{N}$ and a set of arcs $\mathcal{A} \subset \{ (i,j) \,|\, i \in \mathcal{N},\ j \in \mathcal{N} \}$. Each arc joins one node to another. Associated with each arc, there is a *travel time*, which is the time it takes to traverse the arc.

Arcs in the network are *directed*. This means that the arc (i,j) going from i to j is distinct from the arc (j,i), which goes from j to i. In fact, the existence of an arc from i to j has no bearing on whether or not an arc in the opposite direction exists in the network. That is, *one-way* streets are easily modeled in this setup. An example of a network is shown in Figure 1.

A shortest-path-tree problem involves the determination of the shortest paths (routes) to a given node (called the *root* node) from all other nodes in a network.

Because of their special mathematical structure, shortest path problems are amenable to solution by efficient algorithms. In this assignment you will implement *Dijkstra's algorithm* to determine shortest path trees.

Before jumping into a description of Dijkstra's algorithm, we first describe briefly the node and arc classes that we use to represent a network.

2. Node and Arc Classes

The set of all nodes is stored as an array `nodes` of objects from the class `Node`.

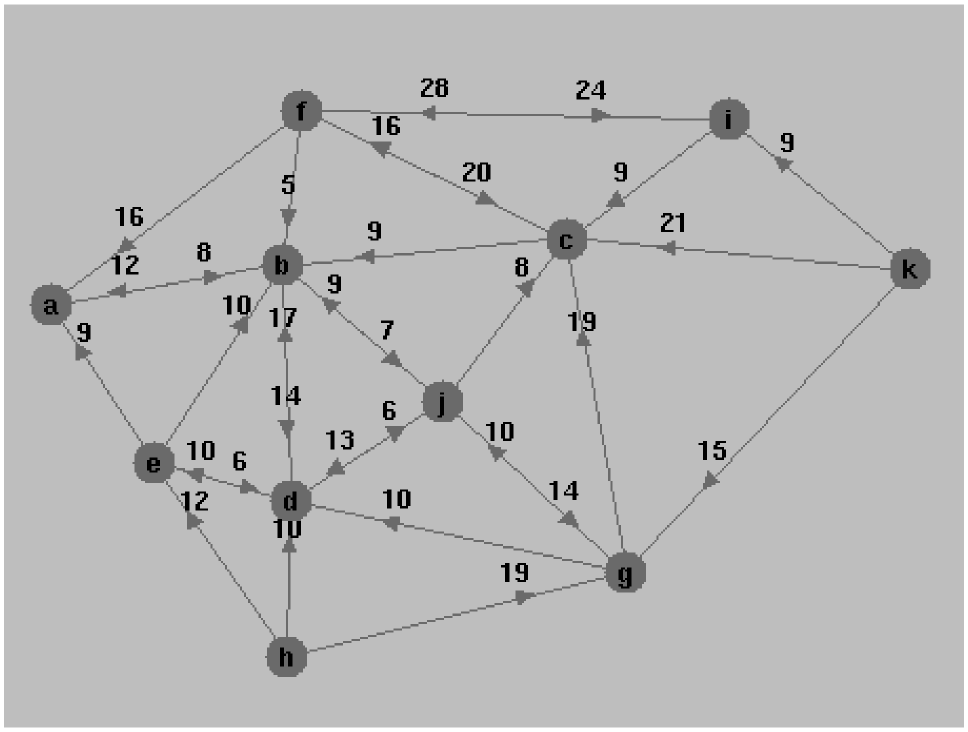

FIGURE 1. Example Network for the Shortest Path Problem

The set of all arcs is stored as an array `arcs` of objects from the class `Arc`.

Node objects contain the following generic information:

- `xcoord` and `ycoord`—spatial position.
- `deg`—number of arcs having their head at the node.
- `arcs`—array of arcs having their head at the node.
- `name`—a string storing the name of the node.

In addition, they contain further information that is specific to Dijkstra's algorithm:

- `parent`—the first arc on a shortest path to the root.
- `pathlen`—the length of time from the current node to the root along the shortest path.
- `finished`—a flag indicating if the node's been added to the shortest path yet or not.
- `next`—a pointer to another node to be used to make a linked list of unfinished nodes.

Arc objects contain generic information:

- `tail`—the node at the *beginning* of the arc.
- `head`—the node at the *end* of the arc.
- `time`—the time required to traverse the arc.

In addition, they contain one extra field that is specific to representing a path tree:

- `treeflag`—a boolean variable indicating whether or not this arc is on the path.

3. Dijkstra's Algorithm

Dijkstra's algorithm computes the shortest path to a given node, called the *root* node, from every other node in the network. The set of paths so found forms a subgraph of the original network. This subgraph turns out to be a tree (a *tree* is a graph having no loops in it). Hence, we can say that Dijkstra's algorithm finds the shortest-path-tree.

Consider the network $\mathcal{G} = \{\mathcal{N}, \mathcal{A}\}$, where $\mathcal{N}$ is the set of nodes (there are a total of n nodes) and $\mathcal{A}$ is the set of arcs. Let t_{ij} denote the time to traverse arc $(i, j) \in \mathcal{A}$.

Dijkstra's algorithm is an iterative procedure. At each iteration, one new node is added to the shortest-path-tree. The process starts with no arcs in the shortest-path-tree and with the root node as the only node in the tree.

At each iteration, the `Node` class member, `path_len`, will contain the time for the quickest path from each node to the root node subject to the constraint that all nodes in the connecting path belong to the path-tree constructed so far.

Initially, `path_len` for the root node is set to 0, `path_len` for nodes that are only one arc removed from the root is set to the traversal time for this arc, and all other nodes have `path_len` set to a huge value (a computer's approximation to infinity).

Furthermore, the `parent` field is initially set to `null` for all nodes except for those that are one arc away from the root. For these nodes, `parent` is set to this connecting arc. Consider a specific node. As the algorithm progresses, the field `parent` for this node will indicate the node in the shortest-path-tree to which this node has to be connected in order to obtain a path of length `path_len` to the root from this node.

While there remain nodes not included in the shortest-path-tree, the following steps are performed:

- Loop through the nodes not in the shortest-path-tree to find the node with the smallest value of `path_len`. Call this node `new_node`. If its `path_len` is infinite, then the network is disconnected and the algorithm must quit here.
- Add `new_node` to the shortest-path-tree and add an arc to the shortest-path-tree linking `new_node` to its parent.
- Update `path_len` and `parent` for the neighbors of `new_node` that still need to be added to the tree. To do this update, for each neighbor `nbr` of `new-node` that is not in the shortest-path-tree, compute the amount of travel time needed to get to the root node from this node assuming that the first leg takes us from `nbr` to `new_node` and then using the shortest path to `root` from `new_node`. If this is path is quicker than the old value of

`path_len` stored for `nbr`, then update `nbr`'s value of `path_len` to this shorter time and change `nbr`'s value of `parent` to `new_node`.

After the shortest-path-tree has been generated, it is easy to find the shortest distance from any node `j` to the root node by walking through the array `parent[]`, starting at node `j`, and adding the distance between each node and its parent stopping once the root node is reached.

4. Linked List of Unfinished Nodes

You should maintain a linked list of unfinished nodes. The field `next` in the `Node` class can be used for this purpose. Initially, the linked list should link everything together:

$$a \to b \to c \to \cdots \to z$$

When selecting a node to declare finished, you should loop just through the linked list. In this way, you avoid looping through nodes that you know are already done. When a node is finished, it should be removed from the linked list.

There is a much more efficient data structure, called a *heap*, that can be used to pick the smallest value from a bunch of numbers. Heaps are covered in more advanced courses such as COS 226.

5. Getting Started

As usual, you need to create a directory in which to work: Begin like this:

```
cd public_html/JAVA/ORF201
mkdir shortpaths
cd shortpaths
```

Then you need to copy over the following files:

```
cp /u/orf201/public_html/JAVA/ORF201/shortpaths/index.html .
cp /u/orf201/public_html/JAVA/ORF201/shortpaths/shortpaths.html .
cp /u/orf201/public_html/JAVA/ORF201/shortpaths/ShortPaths.java .
```

Compile the java code that you just copied over:

```
javac ShortPaths.java
```

Then use appletviewer to see what the code currently does:

```
appletviewer shortpaths.html
```

An applet window should pop up with button labelled *Go Wandering*. Pressing that button brings up another window that allows you to define and then solve shortest path problems. More details on how to use this new window can be found in `shortpaths.html`. All of the user-inteface code is included in the file `ShortPaths.java`. The only thing that is missing is the guts of the method called `dijkstra`. Your assignment is to fill in that method.

6. SHORTPATHS.HTML

Don't forget to edit `shortpaths.html` to make the following changes:

- *Change* `codebase` *from* `"../.."` *to* `http://www.princeton.edu/∼yourname/JAVA`.
- *Add the honor code pledge:*
  This program represents my own work in accordance with University regulations.

7. GOING PUBLIC

If your umask is set for restricted access (i.e., 077), don't forget to make your files public:

```
chmod a+rx public_html/JAVA/ORF201/shortpaths
chmod a+r public_html/JAVA/ORF201/shortpaths/index.html
chmod a+r public_html/JAVA/ORF201/shortpaths/shortpaths.html .
chmod a+r public_html/JAVA/ORF201/shortpaths/ShortPaths.class .
```

After you've made your files public check to see if you can bring your applet up in a browser. Fire up netscape and go to the following address:

```
http://www.princeton.edu/˜yourname/JAVA/ORF201/shortpaths/shortpaths.html
```

If your code works and permissions are set correctly, you should see the shortest paths applet in the browser.

8. MINIMUM REQUIREMENTS

At a minimum:

(1) Your program must correctly implement Dijkstra's algorithm.

9. Possible Enhancements (How to Spend Summer Vacation)

We list here some obvious enhancements to the program. You are not expected to do these things. They are listed here simply for your consideration should you find you have some free time over the summer and want to keep active with your Java programming.

- Change the class `Node` so that the array of arcs having their heads in the node is a linked list of arcs.
- Etc.