Implementation of Graphs Using java.util
Part Two: Weighted Graphs, Spanning Trees, and Shortest Paths

Nicholas J. De Lillo
Nicholas J. De Lillo is Professor of Mathematics and Computer Science at Manhattan College where he has taught courses in computer science, computer engineering, and software engineering at both the undergraduate and graduate levels for over thirty years. In addition, Professor De Lillo regularly teaches courses in the masters program in computer science here at Pace. He is also on the Editorial Board of Technical Reports.

Professor De Lillo is the author of numerous research papers and textbooks in mathematics and computer science. The texts include Advanced Calculus with Applications (1982); Computability with Pascal, co-authored with John S. Mallozzi (1984); A First Course in Computer Science with Ada (1993); Data Structures with C++, co-authored with John S. Mallozzi (1997); Object-Oriented Design in C++ Using the Standard Template Library (2002); and Object-Oriented Design in Java Using java.util (2004).

Professor De Lillo holds a B.S. in mathematics from Manhattan College, an M.A. in mathematics from Fordham University, and the Ph.D. in mathematics from New York University, where he was a student of Martin Davis's.
IMPLEMENTATION OF GRAPHS USING java.util
Part Two: Weighted Graphs, Spanning Trees, and Shortest Paths

Nicholas J. De Lillo
Department of Mathematics and Computer Science
Manhattan College
Riverdale, New York 10471

Abstract

This paper continues the discussion of the implementation of graphs using Java 5.0 begun in [1], with special emphasis on weighted graphs, both directed and undirected, as well as the treatment of minimal spanning trees and shortest paths. Again, the emphasis will be on implementing a number of the key results in this regard, using the predefined List interface and the LinkedList implementation class from the Java Collections hierarchy.

1. Weighted Graphs.

Let $G$ be a digraph. If each edge of $G$ has a third component whose value is a nonnegative integer, called the cost or weight of that edge, the resulting digraph is called a weighted digraph. The following is an example of a weighted digraph, where the weight $w$ of each edge is represented as the third component of each member of $E$, the set of weighted edges of the weighted digraph.

**Example 1:** Suppose we define a weighted digraph with vertices $V = \{ A_0, A_1, A_2, A_3, A_4 \}$ and edges $E = \{ (A_0, A_1, 2), (A_0, A_4, 1), (A_1, A_2, 4), (A_2, A_0, 2), (A_2, A_1, 1), (A_4, A_0, 2) \}$. We may illustrate this using (Figure 1).

![Diagram of a weighted digraph with vertices $A_0, A_1, A_2, A_3, A_4$ and edges with weights.]

(Figure 1)
Weighted digraphs have a number of important applications. For example, these may be used to display the mileage between cities, or the travel time from one point to another, or the cost of executing a move in a game of chess. Let $\mathcal{P}$ be a path from one vertex $u$ of $G$ to another vertex $v$ of $G$. Then the weight of the path from $u$ to $v$ is the sum of the weights of any set of weighted edges that begin at $u$ and terminate at $v$. For example, if we consider the weighted digraph described in (Figure 11.14) with $u = A_0$ and $v = A_2$, then the weight of the path from $u$ to $v$ may be given by the sum of the weights of the edges $(u, A_1, 2) = (A_0, A_1, 2), (A_1, A_2, 4) = (A_1, v, 4)$ or 6. However, there is another path from $u = A_0$ to $v = A_2$ given by $(u, A_1, 2) = (A_0, A_1, 2), (A_1, A_2, 4), (A_2, A_2, 1) = (A_2, v, 1)$ with weight 7, and another path given by $(u, A_1, 2) = (A_0, A_1, 2), (A_1, A_2, 4), (A_2, A_0, 2), (A_0, A_4, 1), (A_4, A_0, 2), (A_0, A_1, 2), (A_1, A_2, 4) = (A_1, v, 4)$ with weight 17. Clearly, there are many other (indeed, in this case, an infinite number of) weighted paths from $u = A_0$ to $v = A_2$.

An important and related practical problem involving weighted digraphs to solve is to determine the path between two vertices with the smallest weight. This is commonly called the Shortest Path Problem, whose solution we will discuss later in this chapter, and is given by an algorithm due to E. W. Dijkstra.\(^1\)

For the moment, we consider the problem of how to represent a weighted digraph internally. Here we use very natural extensions of the internal representation of digraphs using adjacency matrices and adjacency lists. An adjacency matrix representation of a weighted digraph is given by a two-dimensional array of values $w[i, j]$, where

$$w[i, j] = \begin{cases} 
\text{weight of the directed edge beginning at } A_i \text{ and terminating at } A_j, & \text{if such a directed edge exists;} \\
\infty, & \text{if no such edge exists.}
\end{cases}$$

Thus the weighted digraph of (Figure 1) has the associated adjacency matrix

(column index)  
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∞</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>1</td>
<td>∞</td>
<td>∞</td>
<td>4</td>
<td>∞</td>
</tr>
</tbody>
</table>

(row index)  
| 2 | 2 | ∞ | 1 | ∞ | ∞ |

| 3 | ∞ | ∞ | ∞ | ∞ | ∞ |

| 4 | 2 | ∞ | ∞ | ∞ | ∞ | ∞ |

(Figure 2)

Similarly, the adjacency list representation of a weighted digraph \( G = (A_0, A_1, \ldots, A_{n-1}) \) is a one-dimensional array of \( n \) components, each of which is a reference to a linearly linked list or to null. Each of the nodes on the list contains an additional component defining the weight of the directed edge. For example, if \( G \) contains the weighted edge \((A_i, A_j, w)\), then component \( i \) will contain a reference to a node given by

![reference to another node with the same internal structure or null]

To illustrate this, the adjacency list of the weighted digraph of (Figure 1) is given by
How may we implement this in Java using an object-oriented design? We will define a new class called `weightedDigraph`, whose objects are weighted digraphs whose design follows the above adjacency list representation.

2. Implementation Details for Weighted Digraphs.

Following Section 7 of De Lillo, we will describe an implementation of weighted digraphs in Java using adjacency lists. To this end, we first describe a vertex of a weighted digraph as an object of the `weightedVertex` class, defined as

```java
class weightedVertex
{
   // Instance variables.
   private int dest;
   private int weight;
   // Constructor. Constructs a weighted vertex object whose
   // destination vertex is given by value of the first parameter
   // and whose weight is given by the value of the second
   // parameter.
   public weightedVertex(int d, int wt)
   {
      dest = d;
      weight = wt;
   }
```

---

// terminates text of constructor.
// Instance methods.

// Retrieves weight of weighted vertex
public int getWeight()
{
    return weight;
} // terminates text of getWeight()

// Retrieves value of destination vertex
public int getVertex()
{
    return dest;
} // terminates text of getVertex()

// terminates text of weightedVertex class.

This implementation follows the description given in Section 1 of this paper, where each node in the list is viewed as containing an info component of the weightedVertex type. We also define an exception class used for handling exceptions that may arise using certain instance methods defined within the weightedDigraph class. To this end, we include the weightedDigraphException class, whose code is given by

// weightedDigraphException class
class weightedDigraphException extends RuntimeException
{
    // Constructor
    public weightedDigraphException(String str)
    {
        super(str);
    } // terminates text of constructor.
} // terminates text of weightedDigraphException class.

We continue the description of the implementation details by describing an interface for the weightedDigraph class. Again, observe that most of the instance methods described in this interface are the counterparts for weighted digraphs of corresponding instance methods of the Digraph class.

public interface weightedDigraphInterface
{
    // Tests whether the current weighted digraph is empty.
    // Returns true if so, false if not.
    public boolean isEmpty();

    // Returns the number of distinct vertices in the
    // current weighted digraph.
    public int size();

    // Retrieves sum of the weights of all of the edges of
    // the current weighted digraph.
    public int totalWeight();

    // Returns whether v is joined to w by a weighted edge.
    // Returns true if so, false if not.
    public boolean isAdjacent(weightedVertex v, weightedVertex w);
We now describe the design of the instance methods and a description of the necessary instance variables of the implementation of `WeightedDigraphInterface`. The instance variables, with two notable exceptions, are similar to those defined for the `Digraph` class.

// Gives the maximum possible number of vertices in any // weighted digraph.
protected int ArraySize = 10;

// Number of vertices in the current weighted digraph.
protected int currSize;

// Holds the total weight of all of the edges of the current // weighted graph.
protected int currWeight;

// Array holding the weighted vertices of the current // weighted digraph.
protected boolean weightedVertices[] = new boolean[ArraySize];

// Tests whether the current vertex is visited in some traversal.
protected boolean isVisited[] = new boolean[ArraySize];

// Holds weighted edges in the current weighted digraph.
protected LinkedList adjList[] =
    new LinkedList[ArraySize];

// Holds vertices in the current weighted digraph.
protected Set<Integer> vertexSet = new TreeSet<Integer>();

The vertexSet instance variable holds a number of Integer values, none of which are repeated. All we are concerned with here is that every vertex is accounted for, so long as it appears in the weighted digraph.

The constructor constructs an initially empty Digraph object. This is represented in the form of an empty adjacency list, with initial size and total weight equal to zero, and containing no weighted vertices. As a result, each component of the weightedVertices and isVisited arrays is initialized as false. The formal code for the constructor is given by

```
// Constructor. Constructs empty weightedDigraph object
// represented by an empty adjacency list, with current
// size and total weight zero, and with no vertices.
// Hence, each component of the weightedVertices and
// isVisited arrays is initialized as false.
public weightedDigraph()
{
    currSize = 0;
    currWeight = 0;
    for(int index = 0; index < ArraySize; ++index)
    {
        weightedVertices[index]= false;
        isVisited[index] = false;
        adjList[index] = new LinkedList();
    } // terminates text of for-loop
} // terminates text of constructor.
```

The instance methods isEmpty() and size() are defined and coded in exactly the same way as in Digraph. The method totalWeight() applies exclusively to weighted digraphs, and is implemented as

```
// Retrieves sum of the weights of all of the edges of
// the current weighted digraph.
public int totalWeight()
{
    return currWeight;
} // terminates text of totalWeight().
```

The coding of the insertVertex method for weighted digraphs is the direct analog of its counterpart for (unweighted) digraphs. All that is done is the boolean value of the appropriate component of the weightedVertices array is changed from false to true. Otherwise, a weightedDigraphException exception is thrown.

```
// Inserts a new vertex if that vertex is not already present
// and raises an exception if no vertex is inserted, since it is
// already a vertex of the current weighted digraph.
public void insertVertex(weightedVertex p)
{
    // Vertex is not in present digraph.
```
// Perform legitimate insert operation.
if (!weightedVertices[p.getVertex()])
{
  ++currSize;
  weightedVertices[p.getVertex()] = true;
  vertexSet.add(new Integer(p.getVertex()));
} // terminates text of if-clause
else if ((currSize < ArraySize) && weightedVertices[p.getVertex()])
  // Vector is in current digraph. Throw an exception.
  throw new weightedDigraphException("Vertex already in weighed digraph");
else // overflow
  throw new weightedDigraphException("Overflow -- weighted digraph is full");
} // terminates text of insertVertex

The only difference between this version of insertVertex and its counterpart for Digraph is that the subscript of the vertex is computed in this new version by invoking the getVertex() method from the weightedVertex class.

The insertEdge method performs two operations. One is the actual insertion of the weighted edge in the current weighted digraph, assuming that the digraph does not already contain a weighted edge joining these vertices. The second accumulates the total weight of all of the edges by adding the new weight to the existing accumulated weight.

// Inserts a weighted edge from v to w. Throws an exception
// if no edge is constructed.
// Precondition: v, w are vertices of the current weighted digraph, and w is not currently joined to v by a weighted edge.
public void insertEdge(weightedVertex v, weightedVertex w)
{
  // v, w are vertices of the current weighted digraph, and w is not currently joined to v by a weighted edge.
  // Insert new weighted edge and accumulate total weight of weighted digraph.
  if (weightedVertices[v.getVertex()] && weightedVertices[w.getVertex()]
    && !(adjList[v.getVertex()].contains(w)))
  {
    adjList[v.getVertex()].add(w);
    currWeight += w.getWeight();
  } // terminates text of if-clause
  else // if any other condition applies, throw new
    // weightedDigraphException
    throw new weightedDigraphException("Illegal attempt to join weighted edge.");
} // terminates text of insertEdge()

The next method returns a boolean value, and tests whether the weighted vertex v is joined to the weighted vertex w by an edge. The method returns true if so, and false if not.

// Returns whether v is joined to w by a weighted edge.
// Returns true if so, false if not.
public boolean isAdjacent(weightedVertex v, weightedVertex w)
{
    return (adjList[v.getVertex()].contains(w)
        || adjList[w.getVertex()].contains(v));
} // terminates text of isAdjacent().

The next method is the counterpart for weighted digraphs of eraseEdge of (unweighted) digraphs. The key difference now is that weighted edges are involved: the weight of the edge removed is deducted from the total weight of the digraph.

// Removes weighted edge from v to w, if present in the current // weighted digraph.
// Precondition: v, w are vertices in the current weighted digraph, // and there is a weighted edge from v to w.
public void eraseEdge(weightedVertex v, weightedVertex w)
{
    // There is a weighted edge from v to w, and each of v, w // is a weighted vertex in the current weighted digraph.
    if(isAdjacent(v, w) && weightedVertices[v.getVertex()]
        && weightedVertices[w.getVertex()])
    {
        adjList[v.getVertex()].remove(w);
        currWeight -= w.getWeight();
    } // terminates text of if-clause
    else // throw a weightedDigraphException exception
        throw new weightedDigraphException("Edge removal aborted");
} // terminates text of eraseEdge().

The coding of the method eraseVertex() is left as an exercise. It is the analog for weighted digraphs of eraseVertex() for (unweighted) digraphs. The idea is to remove all edges that are incident to the weighted vertex to be removed. This implies that if v is the vertex to be removed, then each weighted edge that is adjacent to v, along with the weight attached to that edge, must be removed. In addition, weightedVertices[v.getVertex()] has to be set equal to false.

The final instance method to be considered is output(), used to output the specifications of the current weighted digraph object. This includes outputting the vertices, edges, and the total weight of the current object.

// Output specifications of current weighted digraph.
public void output()
{
    System.out.println("Vertices are:" + vertexSet);
    //for(int w = 0; w < ArraySize; ++w)
    //if(weightedVertices[w])
    // System.out.print(w +" ");
    // Terminates text of for-loop.
    System.out.println();
    System.out.println("Total weight is " + totalWeight());
    if(isEmpty())
        System.out.println("Current weighted digraph is empty.");
    else
        System.out.println("Current weighted digraph has "

9
    } // Terminates text of output()

If we then execute the code sequence

weightedDigraph wDgraph1 = new weightedDigraph();
weightedVertex a0 = new weightedVertex(0, 0);
wDgraph1.insertVertex(a0);
weightedVertex a1 = new weightedVertex(1, 2);
wDgraph1.insertVertex(a1);
wDgraph1.insertEdge(a0, a1);
weightedVertex a2 = new weightedVertex(2, 4);
wDgraph1.insertVertex(a2);
wDgraph1.insertEdge(a1, a2);
wDgraph1.insertEdge(a2, a0);
weightedVertex a3 = new weightedVertex(0, 2);
weightedVertex a4 = new weightedVertex(1, 4);
weightedVertex a5 = new weightedVertex(4, 2);
wDgraph1.insertVertex(a5);
wDgraph1.insertEdge(a5, a0);
wDgraph1.output();

the output is

Vertices are [0, 1, 2, 4]
Total weight is 8
Current weighted digraph has 4 vertices

The start vertex is 0. Since it is the first vertex inserted, there is no second vertex available to form an edge. We must nevertheless assign a weight component for this vertex which is consistent with maintaining the total weight. Therefore, we assign a weight of zero; consequently, the initial construction of the start vertex is

weightedVertex a0 = new weightedVertex(0, 0);


Let $G = (V, E)$ be an undirected graph. Then the graph $H = (V', E')$ is called a subgraph of $G$ if $V' \subseteq V$ and $E' \subseteq E$, that is, if every vertex of $H$ is also a vertex of $G$ and every edge of $H$ is also an edge of $G$. For example, the graph described in (Figure 4) is a subgraph of the graph of (Figure 5):
A subgraph of $G$ that contains all of the vertices of $G$ and is a tree is called a spanning tree of $G$. For example, suppose $G$ is the connected graph described in (Figure 6(a)):
Then each of the following is a spanning tree for $G$:

(Figure 6(b))

(Figure 6(c))

Finding a spanning tree for a connected graph has important applications, since such trees contain the smallest number of possible edges, and still retain the connection links between all of the vertices of the underlying graph. Specifically, if the graph is connected and has $n$ vertices, then we can show that the spanning tree contains $n - 1$ edges. You may verify this for the graph of (Figure 6(a)) and the spanning trees of (Figure 6(b)) and (Figure 6(c)).
Suppose we turn to weighted connected graphs. These have very important applications, since, for example, such graphs may model maintenance costs between nodes in a communications network, or the travel costs between cities. In the implementation of weighted digraphs discussed in Section 2, we showed how we may accumulate the weight (that is, the cost) for every edge in a weighted digraph. It is not difficult to describe a similar operation for accumulating the weight in an undirected graph. We can then determine a spanning tree that yields a minimum accumulated weight. The spanning tree with the least weight is called a **minimal spanning tree** for that graph.

Suppose \( G = (V, E) \) is a connected weighted graph. How do we find a minimal spanning tree for \( G \)? There are two algorithms that produce a minimal spanning tree: Prim's algorithm and Kruskal's algorithm. We begin by looking at Prim's algorithm.\(^3\) We begin by thinking of \( V \) as subdivided into two disjoint subsets: \( \text{In} \) (for “Inside”) = those elements of \( V \) already included in the minimal spanning tree constructed so far, and \( \text{Out} \) (for “Outside”) = those elements in \( V \) not appearing in the minimal spanning tree constructed so far. Initially, \( \text{In} \) is the set consisting of only the start vertex, and \( \text{Out} \) is the set consisting of the remaining elements of \( V \). The objective is to add new elements to \( \text{In} \) as they are removed from \( \text{Out} \), in succession, by including a new edge joining any element of \( \text{In} \) to an element of \( \text{Out} \) containing the least possible weight. The process terminates when \( \text{In} = V \) and \( \text{Out} = \emptyset \), the empty set. The set \( V \) then contains the vertices of the minimal spanning tree, and the edges that are adjoined at each stage represent the edges of the minimal spanning tree.

Prim's algorithm is an example of a **greedy algorithm**, namely, an algorithm that is based on the selection of a finite collection of objects to join to an initial collection in increments by iteration. Each cycle of the iteration always chooses some object that minimizes some cost function. In the case of Prim's algorithm, the cost function is defined as a function that joins an element of \( \text{In} \) to an element of \( \text{Out} \) by an edge of minimal weight.

**Example 2.** Suppose we consider the weighted graph of (Figure 7):

---

Begin with $\text{In} = \{A_0\}$, $\text{Out} = \{A_1, A_2, A_3, A_4\}$, and $E' =$ current set of edges of the minimal spanning tree $= \emptyset$.

(1) The edges connected to $A_0$: 
- $(A_0, A_1)$ has weight 2,
- $(A_0, A_3)$ has weight 4,
- $(A_0, A_4)$ has weight 3.

Thus, add $A_1$ to $\text{In}$, and add $(A_0, A_1)$ to $E'$. The minimal spanning tree so far consists of

(Figure 8(a))

(2) Add edges connecting $A_1$ to previous edges not selected:

- $(A_1, A_2)$ has weight 4,
- $(A_0, A_3)$ has weight 4,
- $(A_0, A_4)$ has weight 3.

Thus, add $A_4$ to $\text{In}$ and $(A_0, A_4)$ to $E'$. The minimal spanning tree so far consists of
(3) Add edges containing $A_4$ to previous edges not selected:

- $(A_4, A_2)$ has weight 1,
- $(A_1, A_2)$ has weight 4,
- $(A_0, A_3)$ has weight 4.

Thus, add $A_2$ to $I_n$ and $(A_4, A_2)$ to $E'$. The minimal spanning tree so far consists of

(Figure 8(c))

(4) Add edges containing $A_2$ to previous edges not selected:

- $(A_1, A_2)$ has weight 4,
- $(A_0, A_3)$ has weight 4.

Thus, add $A_3$ to $I_n$ and $(A_0, A_3)$ to $E'$. Since now $I_n = V$, the desired minimal spanning tree is
Prim’s algorithm may be designed with the connected weighted graph \( G \) as the only parameter. The algorithm begins by constructing an initially empty tree \( \text{minSp}^4 \) and an initially empty priority queue \( \text{prQueue} \). The role of \( \text{minSp} \) will be to keep track of that portion of the minimal spanning tree constructed so far, and \( \text{prQueue} \) will store the edges of the graph in order of increasing weight. In addition, a set of vertices \( \text{In} \) will store the number of vertices of \( G \) that are currently in \( \text{minSp} \). Initially, \( \text{In} \) will consist only of \( A_0 \), the designated start vertex of \( G \). The algorithm may then be written as

\[
\begin{align*}
\text{Initialize } & \text{minSp as empty}; \\
\text{Initialize } & \text{prQueue as empty}; \\
\text{Initialize } & \text{In} = \{A_0\}; \\
\text{Set } & \text{currVertex} = A_0; \\
\text{while}(V \neq \text{In}) & \quad // \ V \text{ is the set of all vertices of } G \\
\{ \\
\text{Insert all edges containing } & \text{currVertex into } \text{prQueue}; \\
\text{// This insertion may be done using the for-loop} & \\
\text{// for(all edges } e & \text{ containing currVertex)} \\
\text{// prQueue.add(e);} & \\
\text{// Then choose edge at front of current } & \text{prQueue} \\
\text{e} & = \text{prQueue.front();} \\
\text{// Remove edge from } & \text{prQueue} \\
\text{prQueue.remove();} & \\
\text{// Insert } e & \text{ as new edge of } \text{minSp} \\
\text{minSp.insert(e);} & \\
\text{// Add currVertex to } \text{In} & \\
\text{In.add(currVertex);} & \\
\} & \quad // \text{terminates text of while-loop.} \\
\text{// At this point } V = \text{In and } \text{minSp is the desired}\end{align*}
\]

\(^4\) Note that \( \text{minSp} \) is not necessarily a binary tree.
// minimal spanning tree. Return minSp.
return minSp;

We illustrate the algorithm on the graph of Example 2. Initially, minSp and prQueue are empty and In contains only $A_0$, the start vertex of G. See (Figure 9(a)):

(Figure 9(a))

Processing enters the while-loop for the first time, since $V \neq \text{In}$. Thus, each of $(A_0, A_1), (A_0, A_3), (A_0, A_4)$ is added to prQueue in the order from front to rear as

$$(A_0, A_1) \quad (A_0, A_3) \quad (A_0, A_4)$$

Then $e = (A_0, A_1)$ is removed from prQueue, currVertex is set at $A_1$, $e$ is inserted into minSp, and the value of currVertex is added to In, yielding the results indicated in (Figure 9(b)):

(Figure 9(b))

Now $V \neq \text{In}$ is still true, so processing re-enters the while-loop. Here we add any new edges containing $A_1$ into prQueue. This involves a single edge $(A_1, A_2)$. When this edge is inserted into its proper position, prQueue is read from front to rear as

$$(A_0, A_4) \quad (A_0, A_3) \quad (A_1, A_2)$$
Set \( e = (A_0, A_1) \), remove \( e \) from \( \text{prQueue} \), set \( \text{currVertex} = A_4 \), insert \( e \) into \( \text{minSp} \), and the value of \( \text{currVertex} \) is added to \( \text{In} \), producing the result seen in (Figure 9(c)):

(Figure 9(c))

This completes another cycle of the while-loop. Since \( V \neq \text{In} \) is still true, processing re-enters the while-loop. Since the value of \( \text{currVertex} \) is \( A_4 \), we add all new edges containing \( A_4 \) to \( \text{prQueue} \). This involves the single edge \( (A_4, A_2) \), and thus \( \text{prQueue} \) reads from front to rear as

\[
(A_4, A_2) \quad (A_0, A_3) \quad (A_1, A_2)
\]

Set \( e = (A_4, A_2) \), remove \( e \) from \( \text{prQueue} \), set \( \text{currValue} = A_2 \), insert \( e \) into \( \text{minSp} \), and add \( \text{currValue} \) to \( \text{In} \), producing the results seen in (Figure 9(d)):

(Figure 9(d))

18
This completes the execution of another cycle of the while-loop. Since \( V \neq \text{In} \) still holds, processing re-enters the while-loop. But now since there is no edge in \( G \) joining \( A_2 \) to \( A_3 \), \( \text{prQueue} \) does not change. Instead, \( e \) becomes \((A_0, A_3)\) and is removed from \( \text{prQueue}\), \((A_0, A_3)\) is added to \( \text{minSp} \), and \( \text{currVertex} = A_3 \) is added to \( \text{In} \), producing the results of (Figure 9(e)):

![Diagram of graph with vertices and edges]

(Figure 9(e))

This completes another cycle of the while-loop. At this point, \( V \neq \text{In} \) is false; therefore, control passes to the return of the current version of \( \text{minSp} \). This is the desired minimal spanning tree of \( G \).

To analyze Prim's algorithm, we note that its implementation involves an outer while-loop that executes \( n \) times, where \( n \) is \( V.\text{size}() \), the number of vertices of \( G \). But each such cycle involves an inner for-loop, which executes once for each edge containing the current value of \( \text{currVertex} \). If we assume that there are \( m \) edges, where \( m \leq n - 1 \) since \( G \) is connected, we conclude that Prim's algorithm is of order \( O(m \cdot n) \).

Kruskal's algorithm\(^5\) is another example of a greedy algorithm. It takes a completely different approach in constructing the minimal spanning tree of \( G \) by joining subtrees and edges of \( G \) until two subtrees remain. These are then joined by an edge to form the minimal spanning tree. To illustrate Kruskal's algorithm, we again consider the graph of Example 2, and begin by observing that each vertex of \( G \) may be viewed as a subtree with no edges, as in (Figure 10(a)):

---

Configurations of weighted subtrees are commonly called (weighted) forests. Order the edges of $G$ by inserting all of the edges in a priority queue $\text{prQueue}$ in order of increasing weight (see Figure 10(b)):

(Figure 10(b))

Remove edge $(A_4, A_2)$ from $\text{prQueue}$, creating a new forest as described in (Figure 10(c)):

(Figure 10(c))

with $\text{prQueue}$ now given by

(Figure 10(d))
Remove \((A_0, A_1)\) from \(\text{prQueue}\), creating a new forest

\[\begin{array}{c}
A_4 \\
1 \\
A_2 \\
A_0 \\
A_1 \\
2 \\
A_3
\end{array}\]

\((\text{Figure 10(e)})\)

with \(\text{prQueue}\) given by

\[
\begin{array}{c}
\text{prQueue} \\
(A_0, A_1) \quad (A_0, A_3) \quad (A_1, A_2) \\
\text{front} \quad \text{rear}
\end{array}
\]

\((\text{Figure 10(f)})\)

Remove edge \((A_0, A_4)\) from \(\text{prQueue}\), creating a new forest

\[\begin{array}{c}
A_0 \\
2 \\
A_1 \\
3 \\
A_4 \\
1 \\
A_2
\end{array}\]

\((\text{Figure 10(g)})\)

with accompanying \(\text{prQueue}\)
Now remove edge \((A_0, A_3)\) from \(\text{prQueue}\), creating the forest consisting of the single weighted tree

![Diagram](https://via.placeholder.com/150)

(Figure 10(l))

This is the desired minimal spanning tree for \(G\).

How do we express Kruskal’s algorithm in pseudocode? As in Prim’s algorithm, \(G\) is passed as the only parameter. However, unlike Prim’s algorithm, \(\text{minSp}\) is initialized as the forest whose members are just the elements of \(V\), and \(\text{prQueue}\) is initialized as the members of \(E\), ordered according to increasing weight. The algorithm continues execution by successively removing weighted edges from \(\text{prQueue}\), joining these among the members of \(\text{minSp}\) until \(\text{minSp}\) contains exactly one tree, the desired minimal spanning tree of \(G\). Thus the pseudocode may be expressed as

```
Initialize \(\text{minSp} = V\);
Initialize \(\text{prQueue} = E\), ordered in order of increasing weight;
// Add new edges from \(\text{prQueue}\) to members of \(\text{minSp}\)
// to create a new forest.
for(int index = 1; index <= E.size() && minSp.size() <= V.size() - 1;
    ++index)
    if(prQueue.contains(edge[index]))
        minSp.add(edge[index]);
```
// terminates text of for-loop
// At this point, all of the remaining edges have been joined
// and minSp consists of a single element. This is the
// minimal spanning tree for G. Return this tree.
return minSp;

The add() method inside the loop is invoked as many times as is necessary to join all
of the existing vertices of G without producing a cycle. That is, as many times as is
necessary so as to maintain members of minSp as trees. This operation is of order $O(n)$,
where $n = V.size()$, the number of vertices of G. The construction of the initial version
of prQueue requires a sort operation of order $O(m \log m)$, where $m = E.size()$, the
number of edges of G. Consequently, using the fact that $m \leq n - 1$ (see the Exercises at
the end of this chapter), this yields the following result, estimating the complexity of
Kruskal’s algorithm.

If G is a connected weighted graph with n vertices and m edges, Kruskal’s algorithm
constructs a minimal spanning tree for G, with a complexity of $O(m \log n)$.


A very important and practical application of the theory of weighted graphs involves
finding the shortest path between two vertices. We define the shortest path between two
vertices $A_i$ and $A_j$ of a weighted graph G to be the path with the least weight between $A_i$
and $A_j$. This may not necessarily be the path with the least number of vertices joining $A_i$
and $A_j$.

There is a greedy algorithm, due to Dijkstra, for finding the shortest path. This
algorithm was referred to earlier in this paper. We describe the algorithm here, and refer
to it as the Shortest Path Algorithm. The solution proposed by Dijkstra involves a design
using a pair of nested loops, and is thus of order $O(n^2)$.

The algorithm involves a priority queue prQueue of pairs: the first component is a
character string describing a path beginning with the initial vertex and ending with the
vertex described so far toward the destination vertex. The second component of the pair
is the accumulated weight of the path described by the first component. In addition, the
pairs appearing in the priority queue will be arranged in order of increasing weight.

We illustrate this algorithm using the weighted graph G of (Figure 1). Suppose we
wish to determine the shortest path from $A_0$ to $A_2$. We begin by initializing prQueue by
inserting all edges beginning at $A_0$ as in (Figure 1(a)):
After this initialization, the pair ("A₀A₁", 2) is removed from prQueue, and we iterate over the neighbors of A₁. The only other vertex of an edge from A₁ is A₂, and the weight of that edge is 4. Accumulate the weight of the path A₀A₁A₂ to 6, and insert the pair ("A₀A₁A₂", 6) into prQueue as shown in (Figure 11(b)).

Again remove the pair ("A₀A₄", 3) from the front of prQueue, and iterate over the neighbors of A₄. The vertex A₂ is a neighbor of A₄, and the edge joining A₄ to A₂ has a weight of 1. Accumulate the weight of the path A₀A₄A₂ to 4, and insert ("A₀A₄A₂", 4) into prQueue. See (Figure 11(c)).

Remove ("A₀A₃", 4) from prQueue. Since there is no edge from A₃ to A₂, we do not insert any pair back into prQueue. Since pathCount is zero, the front of the current prQueue, as depicted in (Figure 11(d)) contains the shortest path.
We describe a pseudocode version of Dijkstra’s algorithm. To do so, we must make a number of preliminary observations. First, for ease of computation, we will represent a vertex of the graph by its integer-valued subscript. For example, the vertex $A_0$ will be represented by 0, the vertex $A_1$ by 1, and so on. Secondly, we introduce the int-valued variable `numPaths`, whose current value is the number of distinct paths from the designated source vertex to the destination vertex. Indeed, if we consider the weighted graph of (Figure 7), and if the source vertex is 0 (for $A_0$) and if the destination vertex is 2 (for $A_2$), then `numPaths` is 2.

We also define a `Pair` class whose objects represent pairs of the form $(\text{path, weight})$, where `path` is a `String` variable whose characters are integer-valued digits describing a path from the source vertex to some vertex of the graph, and where `weight` is the integer-valued weight of that path. As an example, the `Path` object `("012", 6)` defines the path $A_0A_1A_2$ with total weight 6 from the weighted graph described in (Figure 7). The formal class definition of `Pair` is

```java
public class Pair {
    // Constructor
    public Pair(String pth, int wt) {
        path = pth;
        weight = wt;
    } // terminates text of constructor
    // Instance methods
    // Retrieve path
    public String getPath() {
        return path;
    } // terminates text of getPath
    // Returns weight of current Pair object
    public int getWeight() {
        return weight;
    } // terminates text of getWeight
    // Instance variables
    private String path;
    private int weight;
} // terminates text of Pair class
```
In addition, we introduce a boolean-valued method `edgeExists` defined for any two vertices \( u, v \) by

\[
\text{edgeExists}(u, v) = \begin{cases} 
\text{true}, & \text{if there is an edge of the graph joining } u, v \\
\text{false}, & \text{if not}. 
\end{cases}
\]

Thus, if we refer to the weighted graph of (Figure 7), `edgeExists(4, 2) = true`, and `edgeExists(3, 2) = false`. We may implement `edgeExists` using either the adjacency matrix representation of the weighted graph (see (Figure 2)) or the adjacency list representation of the weighted graph (see (Figure 3)).

Finally, we define the integer-valued variable `numVertices` as the variable storing the number of vertices of the weighted graph.

The pseudocode for Dijkstra’s algorithm may then be expressed as

```java
public Pair getShortestPath(int source, int destination, int numPaths)
{
    // pathCount stores the number of paths from source to destination
    int pathCount = numPaths;
    // Construct priority queue of Pair objects
    ArrPriorityQueue<Integer> prQueue = new ArrPriorityQueue<Integer>();
    // Initialize prQueue with all Pair objects representing edges
    // joining the designated source vertex to any vertex of the graph
    // connected by an edge.
    for(int index = 0; index < numVertices; ++index)
    {
        if(edgeExists(source, index))
            prQueue.add(Pair object with first component consisting of edge
                        represented by a string from source to index, and
                        second component equal to the weight of that edge);
    }
    // Continue path to destination vertex
    while(pathCount > 0)
    {
        // Retrieve front pair stored in prQueue
        Pair frontPair = (Pair)prQueue.front();
        // Remove that front Pair object
        prQueue.remove();
        for(int index = 0; index < numVertices; ++index)
        {
            if(edgeExists(last vertex of frontPair, index))
            {
                frontPair.path += index converted to String;
                frontPair.weight += getWeight(Pair object with first component last
                                           vertex, index, weight);
            }
        }
    }
}
```

\(^6\) This can be replaced by the predefined `PriorityQueue` implementation of Java 5.0.
--pathCount;
} // terminates text of while-loop

// Retrieve Pair object at front of current prQueue
Pair frontPair = (Pair)prQueue.front();
// Remove any unwanted Pair object occupying position at front
// of current prQueue.
if(pathCount == 0) & last vertex of frontPair != destination)
prQueue.remove();
else // front of prQueue contains desired shortest path
return (Pair)prQueue.front();
// If there is no Pair from source to destination, return null
return null;
} // terminates text of getShortestPath

There is a variant of Dijkstra’s algorithm due to R. W. Floyd,\(^7\) that solves the problem
of finding the shortest path between any two vertices in any graph G whose edges have
positive weights. The idea behind the algorithm is very simple, but its complexity is
O(n^3) for a graph of n vertices, since three nested loops are needed for its
implementation. The algorithm begins by setting up a two-dimensional array whose
components represent the shortest distance of paths joining the vertices given by the row
and column subscripts. Initially, the components of this array have values given by the
components of the corresponding adjacency matrix representation of the weighted graph.
In the case of a component with an initial value of \(\infty\), we instead use a large positive
integer value supported by Java’s primitive int type. The algorithm then proceeds by
searching for shorter distances between each pair of vertices. The resulting array then
contains components representing the shortest distance between any vertices of the
weighted graph.

To give a formal description of the algorithm, we define weight[][] as the array
whose components are the respective weights of the edges of the underlying weighted
graph. We also define shortest[][] as the array whose respective components are the
values of the shortest paths between the associated vertices. Thus, the algorithm may be
expressed as

```
public void allPairsShortestPaths(int[][] weight, int[][] shortest)
{
    // Initialize shortest array
    for(int index1 = 0; index1 < n; ++index1)
        for(int index2 = 0; index2 < n; ++index2)
            shortest[index1][index2] = weight[index1][index2];
    // Find shortest path between any two vertices
    for(int index1 = 0; index1 < n; ++index1)
        for(int index2 = 0; index2 < n; ++index2)
            for(int index3 = 0; index3 < n; ++index3)
                if(shortest[index2][index1] + shortest[index1][index3]  
                  < shortest[index2][index3])
                    shortest[index2][index3] = shortest[index2][index1]  
                                         + shortest[index1][index3];
} // terminates text of allPairsShortestPaths
```

\(^7\) Floyd, R. W., “Algorithm 97: Shortest path,” Communications of the Association for Computing
The complexity of \( O(n^3) \) is a consequence of the three nested for-loops used to compute the shortest path between any two vertices. As an example, if we consider the graph of (Figure 7), \( \text{weight[][]} \) is represented as\(^8\)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10000</td>
<td>2</td>
<td>10000</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>10000</td>
<td>4</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>2</td>
<td>10000</td>
<td>4</td>
<td>10000</td>
<td>10000</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>10000</td>
<td>1</td>
<td>10000</td>
<td>10000</td>
</tr>
</tbody>
</table>

(Figure 12)

The final form of \( \text{shortest[][]} \) is

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

(Figure 13)

---

\(^8\) Here "\(\infty\)" is represented by 10000.
Bibliography


The Ivan G. Seidenberg  
School of Computer Science and Information Systems  
Pace University  
Technical Report Series

EDITORIAL BOARD

Editor:  
Allen Stix, Computer Science, Pace--Westchester

Associate Editors:  
Constance A. Knapp, Information Systems, Pace--Westchester  
Susan M. Merritt, Dean, CSIS--Pace

Members:  
Howard S. Blum, Computer Science, Pace--New York  
Mary F. Courtney, Computer Science, Pace--Westchester  
Nicholas J. DeLillo, Mathematics and Computer Science, Manhattan College  
Fred Grossman, Information Systems; Doctor of Professional Studies, Pace--New York and White Plains  
Fran Goertzel Gustavson, Information Systems, Pace--Westchester  
Joseph F. Malerba, Computer Science, Pace--Westchester  
John S. Mallozzi, Computer Information Sciences, Iona College  
John C. Molluzzo, Information Systems, Pace--New York  
Pauline Mosley, Technology Systems, Pace--New York  
Narayan S. Murthy, Computer Science, Pace--New York  
Catherine Ricardo, Computer Information Sciences, Iona College  
Judith E. Sullivan, CSIS Alumna, MS in CS from Pace--Westchester  
Sylvester Tuohy, Computer Science, Pace--Westchester

The School of Computer Science and Information Systems, through the Technical Report Series, provides members of the community an opportunity to disseminate the results of their research by publishing monographs, working papers, and tutorials. Technical Reports is a place where scholarly striving is respected.

All preprints and recent reprints are requested and accepted. New manuscripts are read by two members of the editorial board; the editor decides upon publication. Authors, please note that production is Xerographic from your submission. Statements of policy and mission may be found in issues #29 (April 1990) and #34 (September 1990).

Please direct submissions as well as requests for single copies to:

Allen Stix  
The Ivan G. Seidenberg School of Computer Science and Information Systems  
Goldstein Academic Center  
Pace University  
861 Bedford Road  
Pleasantville, NY  10570-2799