An Extensive Examination of Data Structures Using C# 2.0

Visual Studio 2005

Scott Mitchell
4GuysFromRolla.com

Update January 2005

Summary: A graph, like a tree, is a collection of nodes and edges, but has no rules dictating the connection among the nodes. In this fifth part of the article series, we'll learn all about graphs, one of the most versatile data structures. (22 printed pages)

Download the DataStructures20.msi sample file.

> **Editor's note** This six-part article series originally appeared on MSDN Online starting in November 2003. In January 2005 it was updated to take advantage of the new data structures and features available with the .NET Framework version 2.0, and C# 2.0. The original articles are still available at http://msdn.microsoft.com/vcsharp/default.aspx?pull=/library/en-us/dv_vstechart/html/datastructures_guide.asp.

> **Note** This article assumes the reader is familiar with C#.

Contents

Introduction

Part 1 and Part 2 of this article series focused on linear data structures—the array, the List, the Queue, the Stack, the Hashtable, and the Dictionary. In Part 3 we began our investigation of trees. Recall that trees consist of a set of *nodes*, where all of the nodes share some connection to other nodes. These connections are referred to as *edges*. As we discussed, there are numerous rules spelling out how these connections can occur. For example, all nodes in a tree except for one—the root—must have precisely one *parent* node, while all nodes can have an arbitrary number of children. These simple rules ensure that, for any tree, the following three statements will hold:

1. Starting from any node, any other node in the tree can be reached. That is, there exists no node that can't be reached through some simple path.

2. There are no *cycles*. A cycle exists when, starting from some node *v*, there is some path that travels through some set of nodes v_1, v_2, ..., v_k that then arrives back at *v*.
3. The number of edges in a tree is precisely one less than the number of nodes.

In Part 3 we focused on *binary trees*, which are a special form of trees. Binary trees are trees whose nodes have at most two children.

In this fifth installment of the article series, we're going to examine *graphs*. Graphs are composed of a set of nodes and edges, just like trees, but with graphs there are no rules for the connections between nodes. With graphs there is no concept of a root node, nor is there a concept of parents and children. Rather, a graph is just a collection of interconnected nodes.

> **Note** Realize that all trees are graphs. A tree is a special case of a graph, one whose nodes are all reachable from some starting node and one that has no cycles.

Figure 1 shows three examples of graphs. Notice that graphs, unlike trees, can have sets of nodes that are disconnected from other sets of nodes. For example, graph (a) has two distinct, unconnected set of nodes. Graphs can also contain cycles. Graph (b) has several cycles. One such is the path from v_1 to v_2 to v_4 and back to v_1. Another one is from v_1 to v_2 to v_3 to v_5 to v_4 and back to v_1. (There are also cycles in graph (a).) Graph (c) does not have any cycles, as one less edge than it does number of nodes, and all nodes are reachable. Therefore, it is a tree.

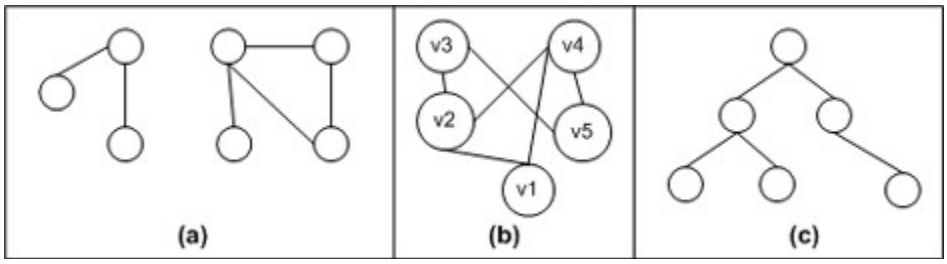


Figure 1. Three examples of graphs

Many real-world problems can be modeled using graphs. For example, search engines model the Internet as a graph, where Web pages are the nodes in the graph and the links among Web pages are the edges. Programs like Microsoft MapPoint that can generate driving directions from one city to another use graphs, modeling cities as nodes in a graph and the roads connecting the cities as edges.

Examining the Different Classes of Edges

Graphs, in their simplest terms, are a collection of nodes and edges, but there are different kinds of edges:

1. Directed versus undirected edges
2. Weighted versus unweighted edges

When talking about using graphs to model a problem, it is usually important to indicate what class of graph you are working with. Is it a graph whose edges are directed and weighted, or one whose edges are undirected and weighted? In the next two sections we'll discuss the differences between directed and undirected edges and weighted and unweighted edges.

Directed and Undirected Edges

The edges of a graph provide the connections between one node and another. By default, an edge is assumed to be bidirectional. That is, if there exists an edge between nodes *v* and *u*, it is assumed that one can travel from *v* to *u* and from *u* to *v*. Graphs with bidirectional edges are said to be *undirected graphs*, because there is no implicit direction in their edges.

For some problems, though, an edge might infer a one-way connection from one node to another. For example, when modeling the Internet as a graph, a hyperlink from Web page *v* linking to Web page *u* would imply that the edge between *v* to *u* would be unidirectional. That is, that one could navigate from *v* to *u*, but not from *u* to *v*. Graphs that use unidirectional edges are said to be *directed graphs*.

When drawing a graph, bidirectional edges are drawn as a straight line, as shown in Figure 1. Unidirectional edges are drawn as an arrow, showing the direction of the edge. Figure 2 shows a directed graph where the nodes are Web pages for a particular Web site and a directed edge from *u* to *v* indicates that there is a hyperlink from Web page *u* to Web page *v*. Notice that both *u* links to *v* and *v* links to *u*, two arrows are used—one from *v* to *u* and another from *u* to *v*.

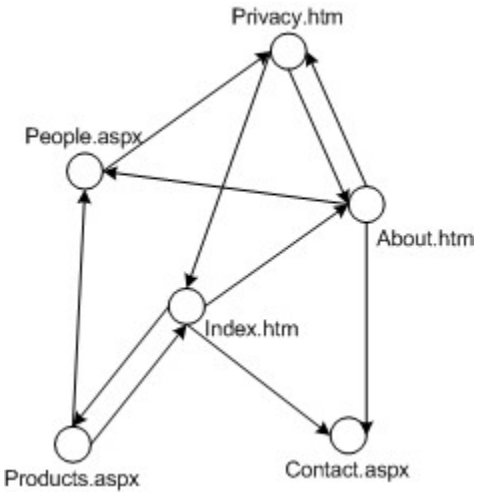


Figure 2. Model of pages making up a website

Weighted and Unweighted Edges

Typically graphs are used to model a collection of "things" and their relationship among these "things." For example, the graph in Figure 2 modeled the pages in a Web site and their hyperlinks. Sometimes, though, it is important to associate some cost with the connection from one node to another.

A map can be easily modeled as a graph, with the cities as nodes and the roads connecting the cities as edges. If we wanted to determine the shortest distance and route from one city to another, we first need to assign a cost from traveling from one city to another. The logical solution would be to give each edge a *weight*, such as how many miles it is from one city to another.

Figure 3 shows a graph that represents several cities in southern California. The cost of any particular path from one city to another is the sum of the costs of the edges along the path. The shortest path, then, would be the path with the least cost. In Figure 3, for example, a trip from San Diego to Santa Barbara is 210 miles if driving through Riverside, then to Barstow, and then back to Santa Barbara. The shortest trip, however, is to drive 100 miles to Los Angeles, and then another 30 up to Santa Barabara.

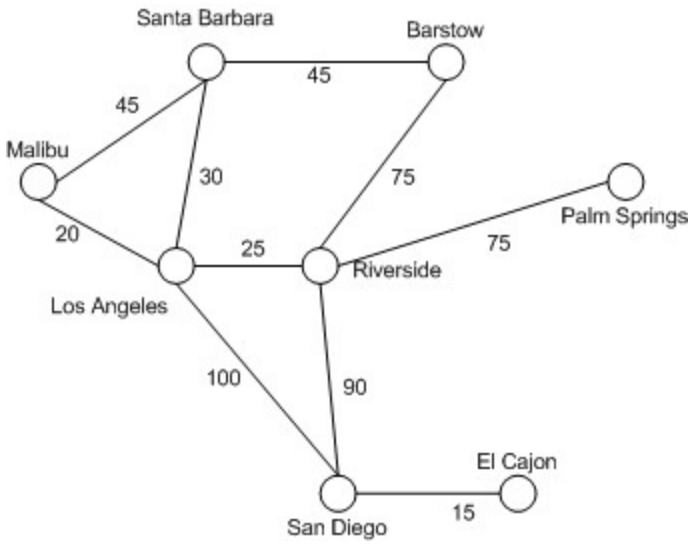


Figure 3. Graph of California cities with edges valued as miles

Realize that directionality and weightedness of edges are orthogonal. That is, a graph can have one of four arrangements of edges:

- Directed, weighted edges
- Directed, unweighted edges
- Undirected, weighted edges
- Undirected, unweighted edges

The graph's in Figure 1 had undirected, unweighted edges. Figure 2 had directed, unweighted edges, and Figure 3 used undirected, weighted edges.

Sparse Graphs and Dense Graphs

While a graph could have zero or a handful of edges, typically a graph will have more edges than it has nodes. What's the maximum number of edges a graph could have, given n nodes? It depends on whether the graph is directed or undirected. If the graph is directed, then each node could have at an edge to every other node. That is, all n nodes could have $n - 1$ edges, giving a total of $n * (n - 1)$ edges, which is nearly n^2.

> **Note** For this article, I am assuming nodes are not allowed to have edges to themselves. In general, though, graphs allow for an edge to exist from a node v back to node v. If self-edges are allowed, the total number of edges for a directed graph would be n^2.

If the graph is undirected, then one node, call it v_1, could have an edge to each and every other node, or $n - 1$ edges. The next node, call it v_2, could have at most $n - 2$ edges, because there already exists an edge from v_2 to v_1. The third node, v_3, could have at most $n - 3$ edges, and so forth. Therefore, for n nodes, there would be at most $(n - 1) + (n - 2) + ... + 1$ edges. Summed up this comes to $[n * (n\text{-}1)] / 2$, or, as you might have already guessed, exactly half as many edges as a directed graph.

If a graph has significantly less than n^2 edges, the graph is said to be *sparse*. For example, a graph with n nodes and n edges, or even $2n$ edges would be said to be sparse. A graph with close to the maximum number of edges is said to be *dense*.

When using graphs in an algorithm it is important to know the ratio between nodes and edges. As we'll see later on in this article, the asymptotic running time operations performed on a graph is typically expressed in terms of the number of nodes and edges in the graph.

Creating a Graph Class

While graphs are a very common data structure used in a wide array of different problems, there is no built-in graph data structure in the .NET Framework. Part of the reason is because an efficient implementation of a `Graph` class depends on a number of factors specific to the problem at hand. For example, graphs are typically modeled in either one of two ways:

- As an adjacency list
- As an adjacency matrix

These two techniques differ in how the nodes and edges of the graph are maintained internally by the `Graph` class. Let's examine both of these approaches and weigh the pros and cons of each approach.

Representing a Graph Using an Adjacency List

In Part 3 we created a base class to represent nodes, the `Node` class. This base class was extended to provide specialized node classes for the `BinaryTree`, `BST`, and `SkipList` classes. Because a each node in a graph has an arbitrary number of neighbors, it might seem plausible that we can simply use the base `Node` class to represent a node in the graph, because the `Node` class consists of a value and an arbitrary number of neighboring `Node` instances. However, while this base class is a step in the right direction, it still lacks needed features, such as a way to associate a cost between neighbors. One option, then, is to create a `GraphNode` class that derives from the base `Node` class and extends it to include the required additional capabilities. Each `GraphNode` class, then, will keep track of its neighboring `GraphNode`s in the base class's `Neighbors` property.

The `Graph` class contains a `NodeList` holding the set of `GraphNode`s that constitute the nodes in the graph. That is, a graph is represented by a set of nodes, and each node maintains a list of its neighbors. Such a representation is called an *adjacency list*, and is depicted graphically in Figure 4.

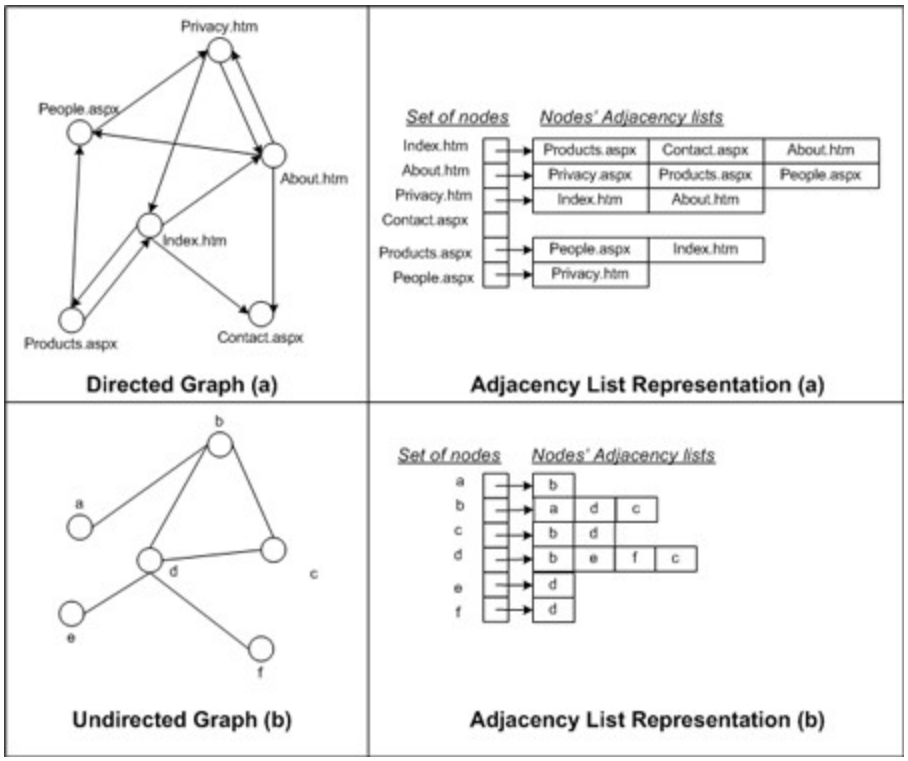


Figure 4. Adjacency list representation in graphical form

Notice that with an undirected graph, an adjacency list representation duplicated the edge information. For example, in adjacency list representation (b) in Figure 4, the node *a* has *b* in its adjacency list, and node *b* also has node *a* in its adjacency list.

Each node has precisely as many `GraphNode`s in its adjacency list as it has neighbors. Therefore, an adjacency list is a very space-efficient representation of a graph—you never store more data than needed. Specifically, for a graph with *V* nodes and *E* edges, a graph using an adjacency list representation will require *V* + *E* `GraphNode` instances for a directed graph and *V* + 2*E* `Node` instances for an undirected graph.

While Figure 4 does not show it, adjacency lists can also be used to represent weighted graphs. The only addition is that for each `GraphNode` *n*'s adjacency list, each `GraphNode` instance in the adjacency list needs to store the cost of the edge from *n*.

The one downside of an adjacency list is that determining if there is an edge from some node *u* to *v* requires that *u*'s adjacency list be searched. For dense graphs, *u* will likely have many `GraphNode`s in its adjacency list. Determining if there is an edge between two nodes, then, takes linear time for dense adjacency list graphs. Fortunately, when using graphs we'll likely not need to determine if there exists an edge between two particular nodes. More often than not, we'll want to simply enumerate *all* the edges of a particular node.

Representing a Graph Using an Adjacency Matrix

An alternative method for representing a graph is to use an *adjacency matrix*. For a graph with *n* nodes, an adjacency matrix is an *n* x *n* two-dimensional array. For weighted graphs the array element (*u*, *v*) would give the cost of the edge between *u* and *v* (or, perhaps -1 if no such edge existed between *u* and *v*. For an unweighted graph, the array could be an array of Booleans, where a True at array element (*u*, *v*) denotes an edge from *u* to *v* and a False denotes a lack of an edge.

Figure 5 depicts how an adjacency matrix representation in graphical form.

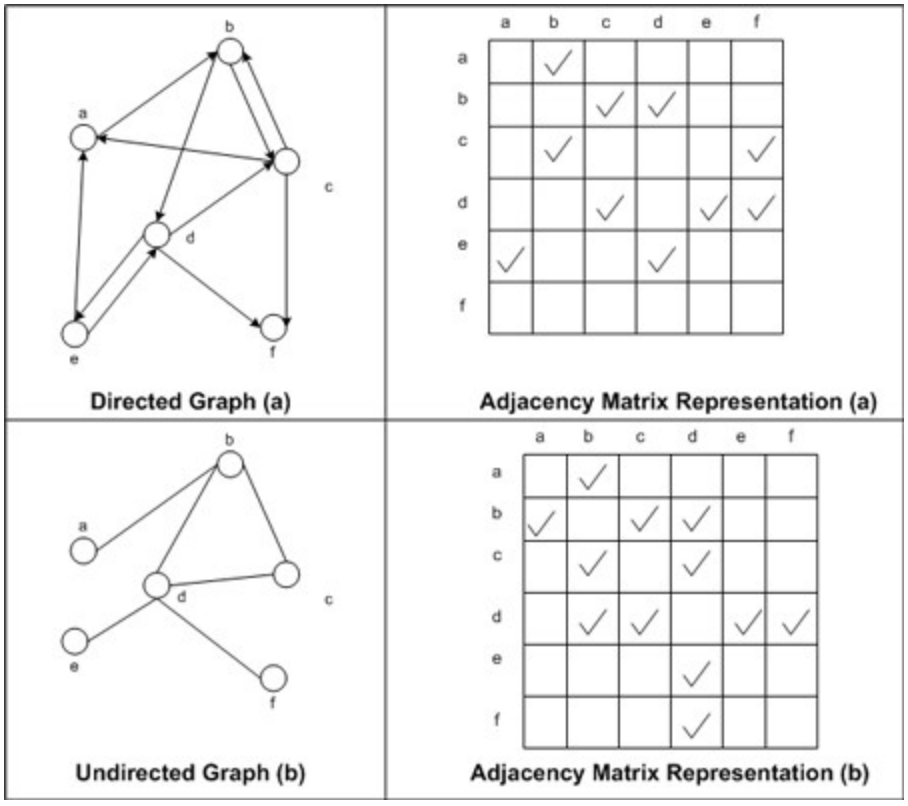


Figure 5. Adjacency matrix representation in graphical form

Note that undirected graphs display symmetry along the adjacency matrix's diagonal. That is, if there is an edge from *u* to *v* in an undirected graph then there will be two corresponding array entries in the adjacency matrix, (*u*, *v*) and (*v*, *u*).

Because determining if an edge exists between two nodes is simply an array lookup, this can be determined in constant time. The downside of adjacency matrices is that they are space inefficient. An adjacency matrix requires an n^2 element array, so for sparse graphs much of the adjacency matrix will be empty. Also, for undirected graphs half of the graph is just repeated information.

While either an adjacency matrix or adjacency list would suffice as an underlying representation of a graph for our `Graph` class, let's move forward using the adjacency list model. I chose this approach primarily because it is a logical extension from the `BinaryTreeNode` and `BinaryTree` classes that we've already created together, and can be implemented by extending the `Node` class used as a base class for the data structures we've examined previously.

Creating the GraphNode Class

The `GraphNode` class represents a single node in the graph, and is derived from the base `Node` class we examined in Part 3 of this article series. The `GraphNode` class extends its base class by providing public access to the `Neighbors` property, as well as providing a `Cost` property. The `Cost` property is of type `List<int>`; for weighted graphs `Cost[i]` it can be used to specify the cost associated with traveling from the `GraphNode` to `Neighbors[i]`.

```
public class GraphNode<T> : Node<T>
{
    private List<int> costs;

    public GraphNode() : base() { }
    public GraphNode(T value) : base(value) { }
    public GraphNode(T value, NodeList<T> neighbors) : base(value, neighbors) { }

    new public NodeList<T> Neighbors
    {
        get
        {
            if (base.Neighbors == null)
                base.Neighbors = new NodeList<T>();

            return base.Neighbors;
        }
    }

    public List<int> Costs
    {
        get
        {
            if (costs == null)
                costs = new List<int>();

            return costs;
        }
    }
}
```

As the code for the `GraphNode` class shows, the class exposes two properties:

- **Neighbors:** this just provides a public property to the protected base class's `Neighbors` property. Recall that Neighbors is of type `NodeList<T>`.
- **Costs:** a `List<int>` mapping a weight from the GraphNode to a specific neighbor.

Building the Graph Class

Recall that with the adjacency list technique, the graph maintains a list of its nodes. Each node, then, maintains a list of adjacent nodes. So, in creating the `Graph` class we need to have a list of `GraphNode`s. This set of nodes is maintained using a `NodeList` instance. (We examined the `NodeList` class in Part 3; this class was used by the `BinaryTree` and `BST` classes, and was extended for the `SkipList` class.) The `Graph` class exposes its set of nodes through the public property `Nodes`.

Additionally, the `Graph` class has a number of methods for adding nodes and directed or undirected and weighted or unweighted edges between nodes. The `AddNode()` method adds a node to the graph, `while AddDirectedEdge()` and `AddUndirectedEdge()` allow a weighted or unweighted edge to be associated between two nodes.

In addition to its methods for adding edges, the `Graph` class has a `Contains()` method that returns a Boolean indicating if a particular value exists in the graph or not. There is also a `Remove()` method that deletes a `GraphNode` and all edges to and from it. The germane code for the `Graph` class is shown below (some of the overloaded methods for adding edges and nodes have been removed for brevity):

```
public class Graph<T> : IEnumerable<T>
{
    private NodeList<T> nodeSet;

    public Graph() : this(null) {}
    public Graph(NodeList<T> nodeSet)
    {
        if (nodeSet == null)
            this.nodeSet = new NodeList<T>();
        else
            this.nodeSet = nodeSet;
    }

    public void AddNode(GraphNode<T> node)
    {
        // adds a node to the graph
        nodeSet.Add(node);
    }

    public void AddNode(T value)
    {
        // adds a node to the graph
        nodeSet.Add(new GraphNode<T>(value));
    }

    public void AddDirectedEdge(GraphNode<T> from, GraphNode<T> to, int cost)
    {
        from.Neighbors.Add(to);
        from.Costs.Add(cost);
    }

    public void AddUndirectedEdge(GraphNode<T> from, GraphNode<T> to, int cost)
    {
        from.Neighbors.Add(to);
        from.Costs.Add(cost);

        to.Neighbors.Add(from);
        to.Costs.Add(cost);
    }

    public bool Contains(T value)
    {
        return nodeSet.FindByValue(value) != null;
    }
```

```
    public bool Remove(T value)
    {
        // first remove the node from the nodeset
        GraphNode<T> nodeToRemove = (GraphNode<T>) nodeSet.FindByValue(value);
        if (nodeToRemove == null)
            // node wasn't found
            return false;

        // otherwise, the node was found
        nodeSet.Remove(nodeToRemove);

        // enumerate through each node in the nodeSet, removing edges to this node
        foreach (GraphNode<T> gnode in nodeSet)
        {
            int index = gnode.Neighbors.IndexOf(nodeToRemove);
            if (index != -1)
            {
                // remove the reference to the node and associated cost
                gnode.Neighbors.RemoveAt(index);
                gnode.Costs.RemoveAt(index);
            }
        }

        return true;
    }

    public NodeList<T> Nodes
    {
        get
        {
            return nodeSet;
        }
    }

    public int Count
    {
        get { return nodeSet.Count; }
    }
}
```

Using the Graph Class

At this point, we have created all of the classes needed for our graph data structure. We'll soon turn our attention to some of the more common graph algorithms, such as constructing a minimum spanning tree and finding the shortest path from a single node to all other nodes, but before we do let's examine how to use the `Graph` class in a C# application.

Once we create an instance of the `Graph` class, the next task is to add the `Node`s to the graph. This involves calling the `Graph` class's `AddNode()` method for each node to add to the graph. Let's recreate the graph from Figure 2. We'll need to start by adding six nodes. For each of these nodes let's have the `Key` be the Web page's filename; we'll

leave the `Data` as `null`, although this might conceivably contain the contents of the file, or a collection of keywords describing the Web page content.

```
Graph<string> web = new Graph<string>();
web.AddNode("Privacy.htm");
web.AddNode("People.aspx");
web.AddNode("About.htm");
web.AddNode("Index.htm");
web.AddNode("Products.aspx");
web.AddNode("Contact.aspx");
```

Next we need to add the edges. Because this is a directed, unweighted graph, we'll use the `Graph` class's `AddDirectedEdge(u, v)` method to add an edge from *u* to *v*.

```
web.AddDirectedEdge("People.aspx", "Privacy.htm");  // People -> Privacy

web.AddDirectedEdge("Privacy.htm", "Index.htm");    // Privacy -> Index
web.AddDirectedEdge("Privacy.htm", "About.htm");    // Privacy -> About

web.AddDirectedEdge("About.htm", "Privacy.htm");    // About -> Privacy
web.AddDirectedEdge("About.htm", "People.aspx");    // About -> People
web.AddDirectedEdge("About.htm", "Contact.aspx");   // About -> Contact

web.AddDirectedEdge("Index.htm", "About.htm");      // Index -> About
web.AddDirectedEdge("Index.htm", "Contact.aspx");   // Index -> Contacts
web.AddDirectedEdge("Index.htm", "Products.aspx");  // Index -> Products

web.AddDirectedEdge("Products.aspx", "Index.htm");  // Products -> Index
web.AddDirectedEdge("Products.aspx", "People.aspx");// Products -> People
```

After these commands, `web` represents the graph shown in Figure 2. Once we have a constructed a graph we'll typically want to answer some questions. For example, for the graph we just created we might want to answer, "What's the least number of links a user must click to reach any Web page when starting from the homepage (`Index.htm`)?" To answer such questions we can usually fall back on using existing graph algorithms. In the next section we'll examine two common algorithms for weighted graphs: constructing a minimum spanning tree and finding the shortest path from one node to all others.

A Look at Some Common Graph Algorithms

Because graphs are a data structure that can be used to model a bevy of real-world problems, there are innumerous algorithms designed to find solutions for common problems. To further our understanding of graphs, let's take a look at two of the most studied applications of graphs: finding a minimum spanning tree and computing the shortest path from a source node to all other nodes.

The Minimum Spanning Tree Problem

Imagine that you work for the phone company and your task is to provide phone lines to a village with 10 houses, each labeled H1 through H10. Specifically this involves running a single cable that connects each home. That is, the cable must run through houses H1, H2, and so forth, up through H10. Due to geographic obstacles—hills, trees, rivers, and so on—it is not feasible to necessarily run the cable from one house to another.

Figure 6 shows this problem depicted as a graph. Each node is a house, and the edges are the means by which one house can be wired up to another. The weights of the edges dictate the distance between the homes. Your task is to wire up all ten houses using the least amount of telephone wiring possible.

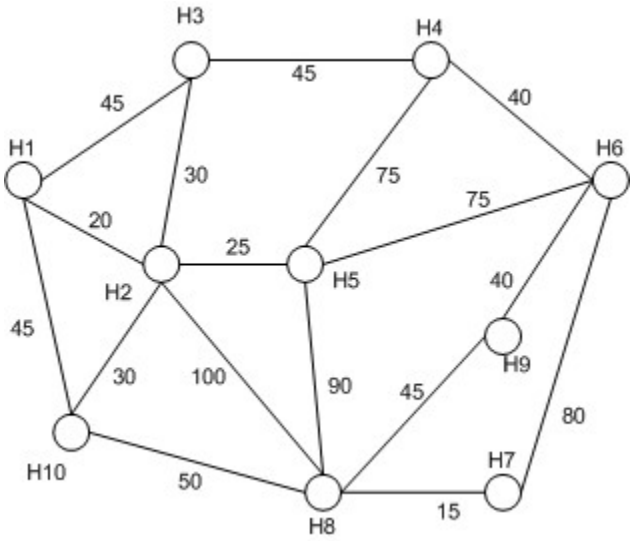


Figure 6. Graphical representation of hooking up a 10-home village with phone lines

For a connected, undirected graph, there exists some subset of the edges that connect all the nodes and does not introduce a cycle. Such a subset of edges would form a tree (because it would comprise one less edge than vertices and is acyclic), and is called a *spanning tree*. There are typically many spanning trees for a given graph. Figure 7 shows two valid spanning trees from the Figure 6 graph. (The edges forming the spanning tree are bolded.)

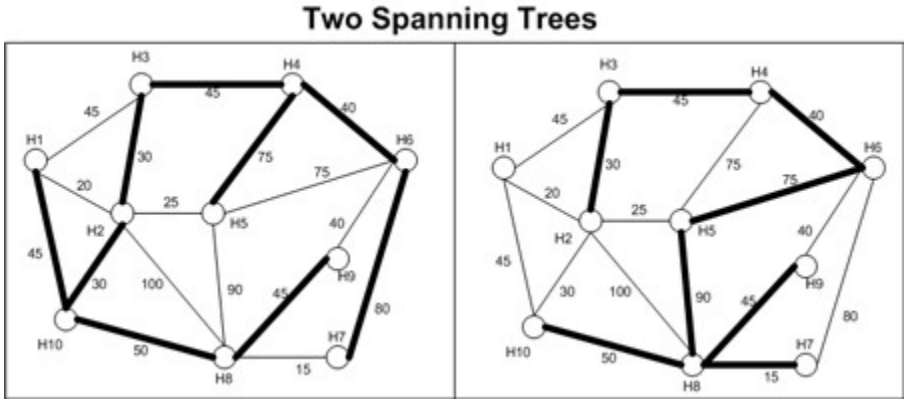


Figure 7.Spanning tree subsets based on Figure 6

For graphs with weighted edges, different spanning trees have different associated costs, where the cost is the sum of the weights of the edges that comprise the spanning tree. A *minimum spanning tree*, then, is the spanning tree with a minimum cost.

There are two basic approaches to solving the minimum spanning tree problem. One approach is build up a spanning tree by choosing the edges with the minimum weight, so long as adding that edge does not create a cycle among the edges chosen thus far. This approach is shown in Figure 8.

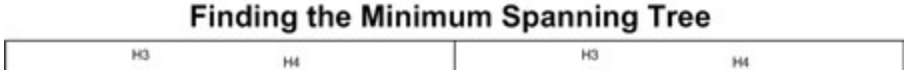


Figure 8. Minimum spanning tree that uses the edges with the minimum weight

The other approach builds up the spanning tree by dividing the nodes of the graph into two disjoint sets: the nodes currently in the spanning tree and those nodes not yet added. At each iteration, the least weighted edge that connects the spanning tree nodes to a node not in the spanning tree is added to the spanning tree. To start off the algorithm, some random start node must be selected. Figure 9 illustrates this approach in action, using H1 as the starting node. (In Figure 9 those nodes that are in the set of nodes in the spanning tree are shaded light yellow.)

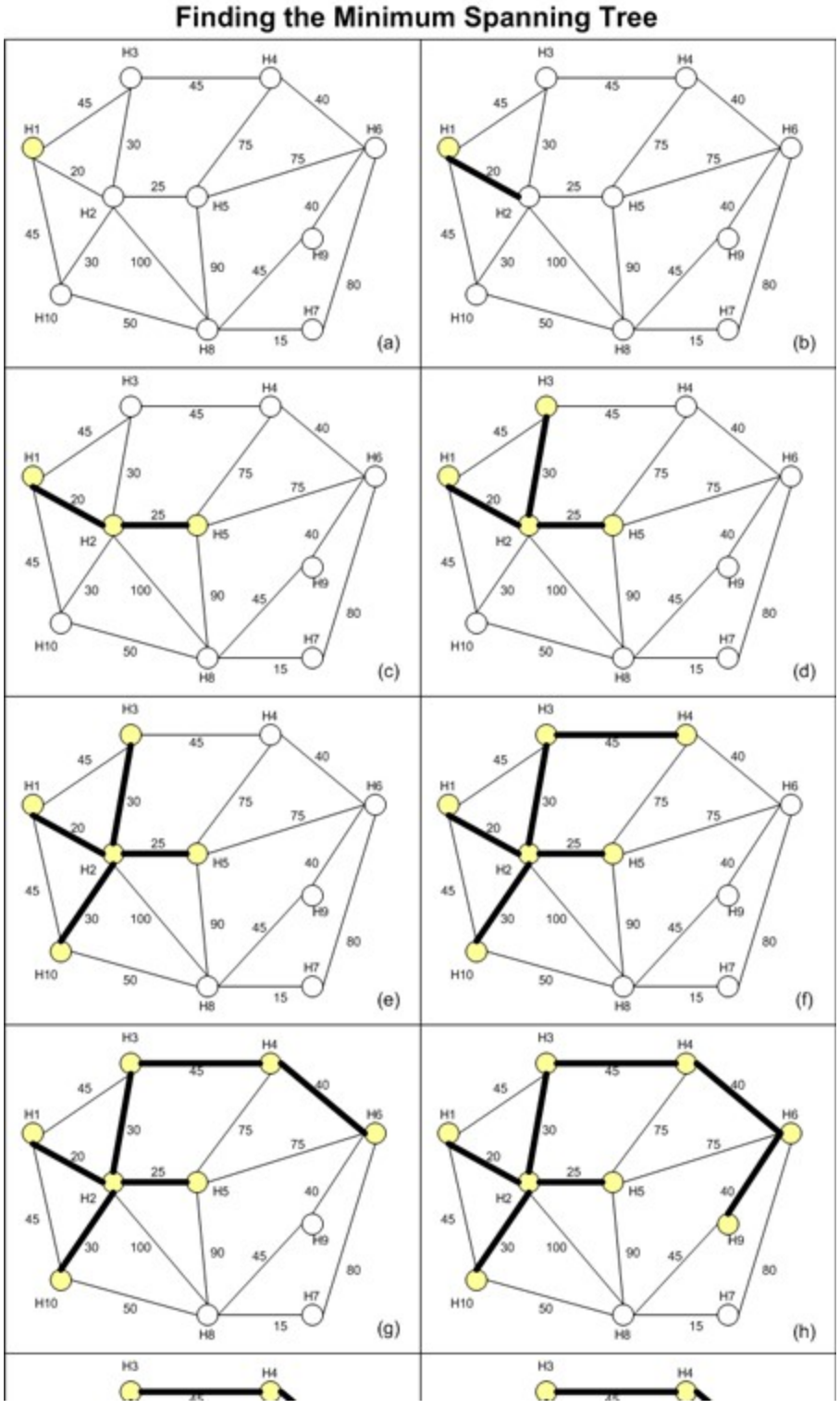


Figure 9. Prim method of finding the minimum spanning tree

Notice that the techniques illustrated in both Figures 8 and 9 arrived at the same minimum spanning tree. If there is only one minimum spanning tree for the graph, then both of these approaches will reach the same conclusion. If, however, there are multiple minimum spanning trees, these two approaches might arrive with different results (both results will be correct, naturally).

> **Note** The first approach we examined was discovered by Joseph Kruskal in 1956 at Bell Labs. The second technique was discovered in 1957 by Robert Prim, also a researcher at Bell Labs. There is a plethora of information on these two algorithms on the Web, including Java applets showing the algorithms in progress graphically (Kruskal's Algorithm | Prim's Algorithm), as well as source code in a variety of languages.

Computing the Shortest Path from a Single Source

When flying from one city to another, part of the headache is finding a route that requires the fewest number of connections—who likes their flight from New York to L.A. to first go from New York to Chicago, then Chicago to Denver, and finally Denver to L.A? Rather, most people would rather have a direct flight straight from New York to L.A.

Imagine, however, that you are not one of those people. Instead, you are someone who values his money much more than his time, and are most interested in finding the *cheapest* route, regardless of the number of connections. This might mean flying from New York to Miami, then Miami to Dallas, then Dallas to Phoenix, Phoenix to San Diego, and finally San Diego to L.A.

We can solve this problem by modeling the available flights and their costs as a directed, weighted graph. Figure 10 shows such a graph.

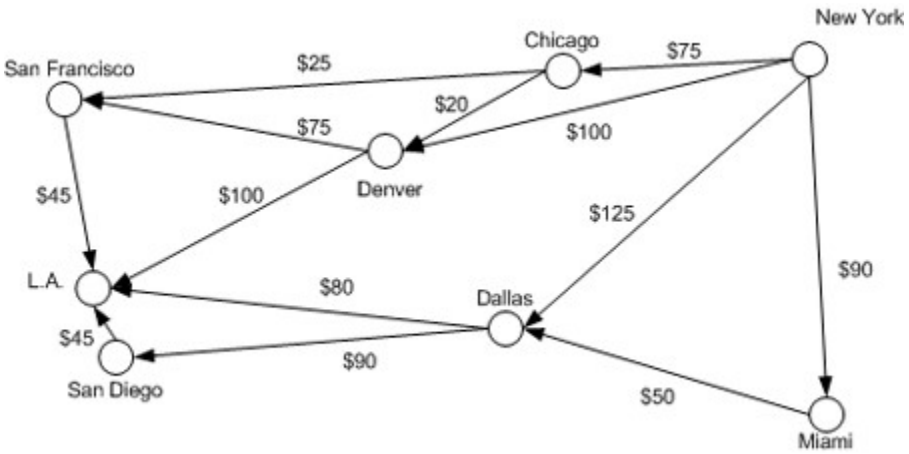


Figure 10. Modeling of available flights based on cost

What we are interested in knowing is what is the least expensive path from New York to L.A. By inspecting the graph, we can quickly determine that it's from New York to Chicago to San Francisco and finally down to L.A., but in order to have a computer accomplish this task we need to formulate an algorithm to solve the problem at hand.

The late Edsger Dijkstra, one of the most noted computer scientists of all time, invented the most commonly used algorithm for finding the shortest path from a source node to all other nodes in a weighted, directed graph. This

algorithm, dubbed Dijkstra's Algorithm, works by maintaining two tables, each of which have a record for each node. These two tables are:

- A distance table, which keeps an up-to-date "best distance" from the source node to every other node.
- A route table, which, for each node *n*, indicates what node was used to reach *n* to get the best distance.

Initially, the distance table has each record set to some high value (like positive infinity) except for the start node, which has a distance to itself of 0. The route table's rows are all set to `null`. Also, a collection of nodes, *Q*, that need to be examined is maintained; initially, this collection contains all of the nodes in the graph.

The algorithm proceeds by selecting (and removing) the node from *Q* that has the lowest value in the distance table. Let this selected node be called *n* and the value in the distance table for *n* be *d*. For each of the *n*'s edges, a check is made to see if *d* plus the cost to get from *n* to that particular neighbor is less than the value for that neighbor in the distance table. If it is, then we've found a better way to reach that neighbor, and the distance and route tables are updated accordingly.

To help clarify this algorithm, let's begin applying it to the graph from Figure 10. Because we want to know the cheapest route from New York to L.A. we use New York as our source node. Our initial distance table, then, contains a value of infinity for each of the other cities, and a value of 0 for New York. The route table contains `null`s for all entries, and *Q* contains all nodes (see Figure 11).

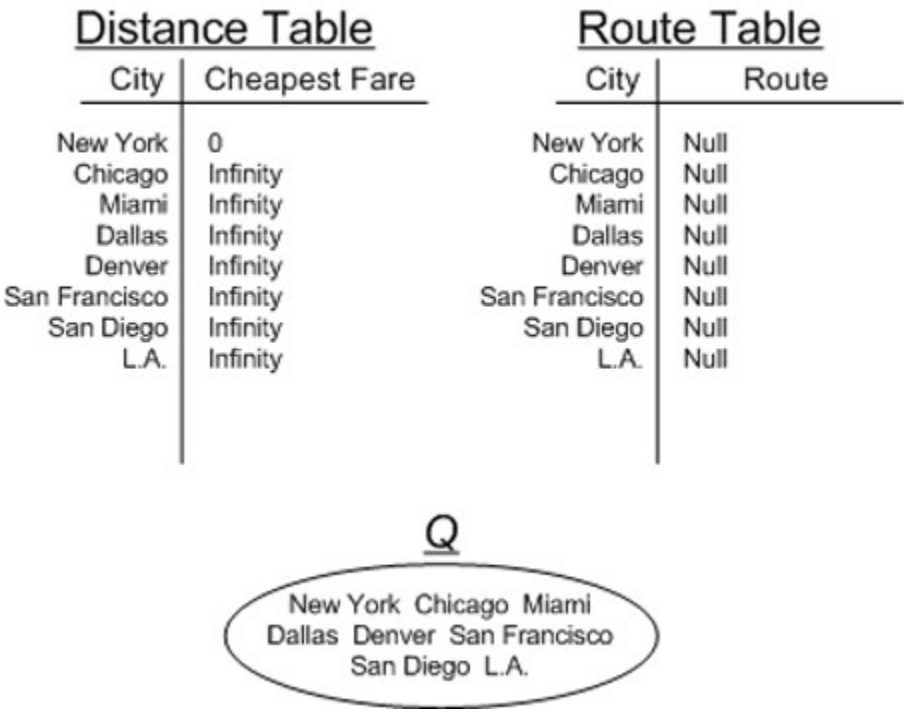


Figure 11. Distance table and route table for determining cheapest fare

We start by extracting the city from *Q* that has the lowest value in the distance table—New York. We then examine each of New York's neighbors and check to see if the cost to fly from New York to that neighbor is less than the best cost we know of, namely the cost in the distance table. After this first check, we'd have removed New York from *Q* and updated the distance and route tables for Chicago, Denver, Miami, and Dallas.

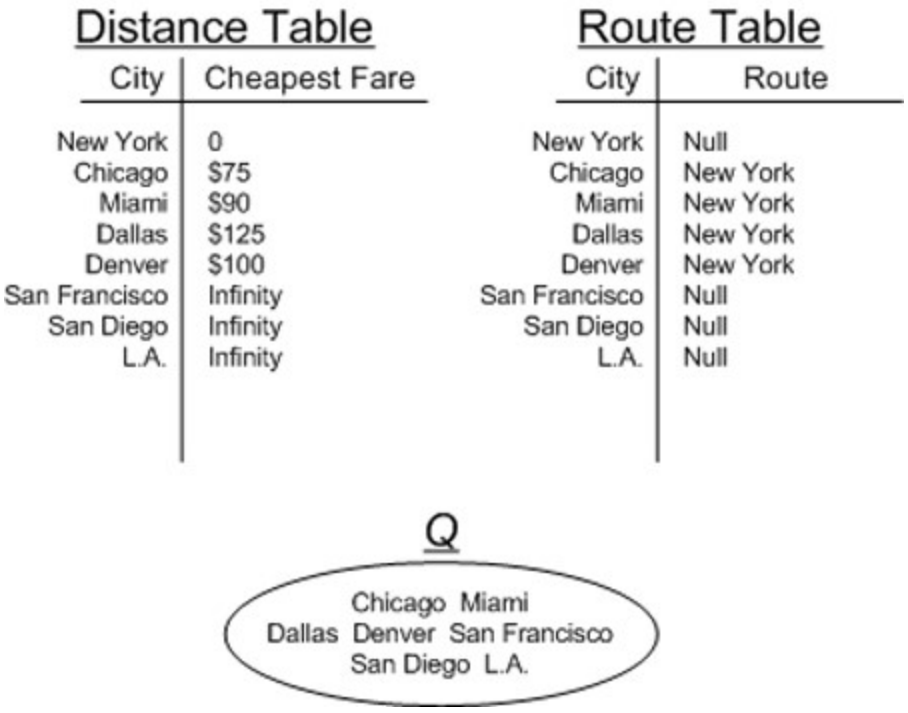


Figure 12. Step 2 in the process of determining the cheapest fare

The next iteration gets the cheapest city out of *Q*, Chicago, and then checks its neighbors to see if there is a better cost. Specifically, we'll check to see if there's a better route for getting to San Francisco or Denver. Clearly the cost to get to San Francisco from Chicago—$75 + $25—is less than Infinity, so San Francisco's records are updated. Also, note that it is cheaper to fly from Chicago to Denver than from New York to Denver ($75 + $20 < $100), so Denver is updated as well. Figure 13 shows the values of the tables and *Q* after Chicago has been processed.

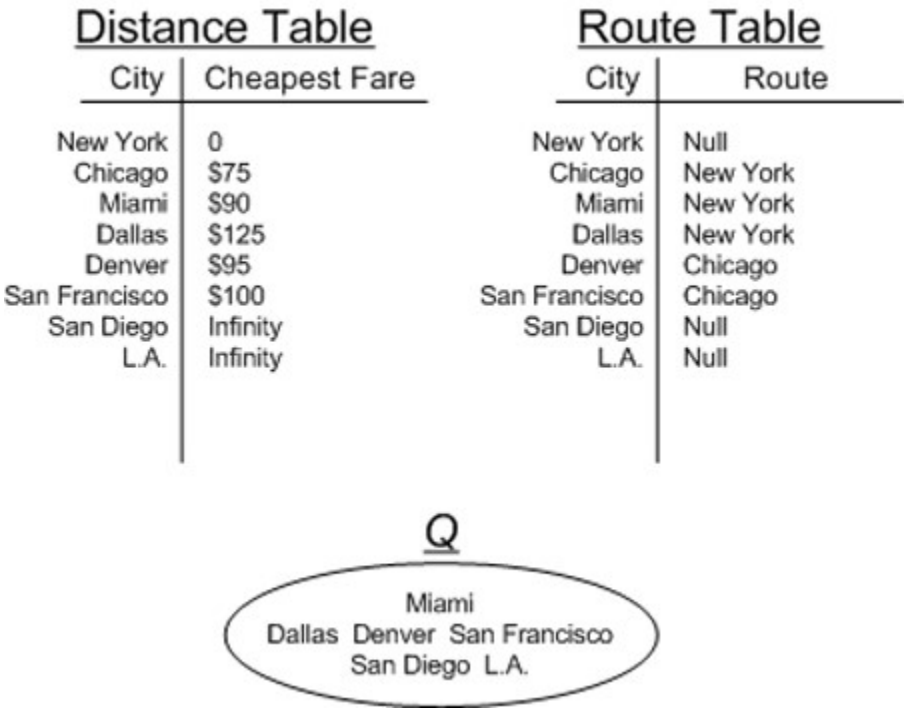
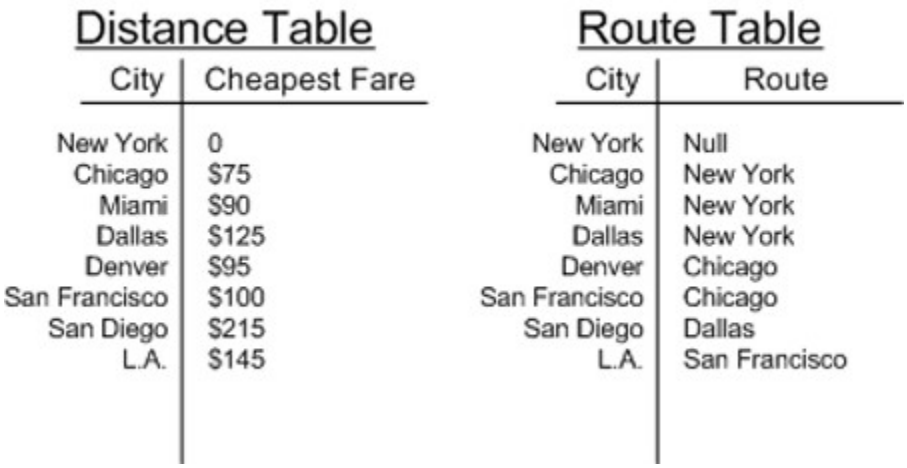


Figure 13. Table status after the third leg of the process is finished

This process continues until there are no more nodes in *Q*. Figure 14 shows the final values of the tables when *Q* has been exhausted.



Distance Table

City	Cheapest Fare
New York	0
Chicago	$75
Miami	$90
Dallas	$125
Denver	$95
San Francisco	$100
San Diego	$215
L.A.	$145

Route Table

City	Route
New York	Null
Chicago	New York
Miami	New York
Dallas	New York
Denver	Chicago
San Francisco	Chicago
San Diego	Dallas
L.A.	San Francisco

Figure 14. Final results of determining the cheapest fare

At the point of exhausting *Q*, the distance table will contain the lowest cost from New York to each city. To determine the flight path to arrive at L.A., start by examining the L.A. entry in the route table and work back up to New York. That is, the route table entry for L.A. is San Francisco, meaning the last leg of the flight to L.A. leaves from San Francisco. The route table entry for San Francisco is Chicago, meaning you'll get to San Francisco via Chicago. Finally, Chicago's route table entry is New York. Putting this together we see that the cheapest flight path is from New York to Chicago to San Francisco to L.A, and costs $145.

> **Note** To see a working implementation of Dijkstra's Algorithm check out the download for this article, which includes a testing application for the `Graph` class that determines the shortest distance from one city to another using Dijkstra's Algorithm.

Conclusion

Graphs are a commonly used data structure because they can be used to model many real-world problems. A graph consists of a set of nodes with an arbitrary number of connections, or edges, between the nodes. These edges can be either directed or undirected and weighted or unweighted.

In this article we examined the basics of graphs and created a `Graph` class. This class was similar to the `BinaryTree` class created in Part 3, the difference being that instead of only have a reference for at most two edges, the `Graph` class's `GraphNode`s could have an arbitrary number of references. This similarity is not surprising because trees are a special case of graphs.

In addition to creating a `Graph` class, we also looked at two common graph algorithms, the minimum spanning tree problem and computing the shortest path from some source node to all other nodes in a weighted, directed graph. While we did not examine source code to implement these algorithms, there are plenty source code examples available on the Internet. Too, the download included with this article contains a testing application for the `Graph` class that uses Dijkstra's Algorithm to compute the shortest route between two cities.

In the next installment, Part 6, we'll look at efficiently maintaining disjoint sets. Disjoint sets are a collection of two or more sets that do not share any elements in common. For example, with Prim's Algorithm for finding the minimum

spanning tree, the nodes of the graph can be divided into two disjoint sets: the set of nodes that currently constitute the spanning tree and the set of nodes that are not yet in the spanning tree.

References

- Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. "Introduction to Algorithms." MIT Press. 1990.

Scott Mitchell, author of six books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies because January 1998. Scott works as an independent consultant, trainer, and writer, and holds a Masters degree in Computer Science from the University of California – San Diego. He can be reached at mitchell@4guysfromrolla.com, or via his blog at http://ScottOnWriting.NET.