CHAPTER 11

Graphs and Graph Algorithms

The study of networks has become one of the great scientific hotbeds of this century, though mathematicians and others have been studying networks for many hundreds of years. Recent developments in computer technology (the Internet, for example) and in social theory (the social network, as popularized by the concept of "six degrees of separation"), not to mention social media, have put a spotlight on the study of networks.

In this chapter we'll look at how networks are modeled with graphs. We'll define what a graph is, how to represent graphs in JavaScript, and how to implement important graph algorithms. We'll also discuss the importance of choosing the correct data representation when working with graphs, since the efficiency of graph algorithms largely depends on the data structure used to represent a graph.

Graph Definitions

A graph consists of a set of *vertices* and a set of *edges*. Think of a map of a US state. Each town is connected with other towns via some type of road. A map is a type of graph where each town is a vertex, and a road that connects two towns is an edge. Edges are defined as a pair (v1, v2), where v1 and v2 are two vertices in a graph. A vertex can also have a weight, which is sometimes called a cost. A graph whose pairs are ordered is called a *directed graph*, or just a *digraph*. When pairs are ordered in a directed graph, an arrow is drawn from one pair to another pair. Directed graphs indicate the flow direction from vertex to vertex. A flowchart that indicates the direction of computations in a computer program is an example of a directed graph. A directed graph is shown in Figure 11-1.

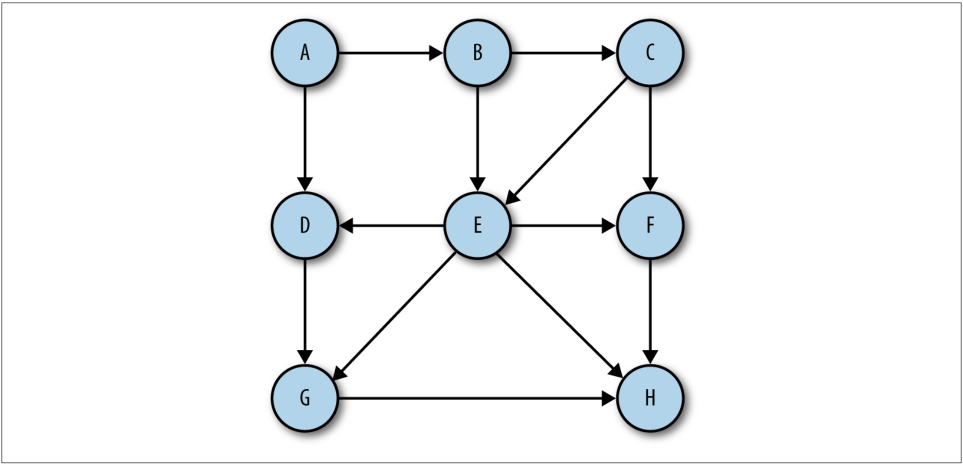

Figure 11-1. A digraph (directed graph)

If a graph is not ordered, it is called an *unordered graph*, or just a graph. An example of an unordered graph is shown in Figure 11-2.

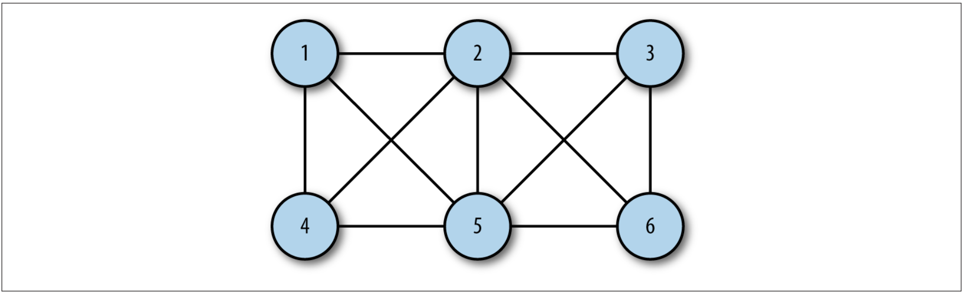

Figure 11-2. An unordered graph

A *path* is a sequence of vertices in a graph such that all vertices in the path are connected by edges. The length of a path is the number of edges from the first vertex in the path to the last vertex. A path can also consist of a vertex to itself, which is called a loop. Loops have a length of 0.

A *cycle* is a path with at least one edge whose first and last vertices are the same. A *simple cycle* is one with no repeated edges or vertices for both directed and undirected graphs. Paths that repeat other vertices besides the first and last vertices are called *general cycles*.

Two vertices are considered *strongly* connected if there is a path from the first vertex to the second vertex, and vice versa. If the graph is a directed graph, and all its vertices are strongly connected, then the directed graph is considered strongly connected.

Real-World Systems Modeled by Graphs

Graphs are used to model many different types of real-world systems. One example is traffic flow. The vertices represent street intersections, and the edges represent the streets. Weighted edges can be used to represent speed limits or the number of lanes. Modelers can use the system to determine the best routes and the streets most likely to suffer from traffic jams.

Any type of transportation system can be modeled using a graph. For example, an airline can model its flight system using a graph. Each airport is a vertex, and each flight from one vertex to another is an edge. A weighted edge can represent the cost of a flight from one airport to another, or perhaps the distance from one airport to another, depending upon what is being modeled.

Computer networks, including local area networks and much broader networks such as the Internet, are also frequently modeled with graphs. Another example of a real-word system that can be modeled by a graph is a consumer market, where vertices represent both institutions (vendors) and consumers.

The Graph Class

At first glance, a graph looks much like a tree or a binary tree, and you might be tempted to try to build a graph class like a tree, using nodes to represent each vertex. There are problems with using an object-based approach like that, however, because graphs can grow quite large. Representing a graph using just objects can quickly become inefficient, so we will look at a different scheme for representing both vertices and edges.

Representing Vertices

The first step in building a `Graph` class is to build a `Vertex` class to store the vertices of a graph. This class has the same duties that the `Node` class had with linked lists and binary search trees. The `Vertex` class needs two data members: one for identifying the vertex and the other to store a Boolean value indicating whether or not the vertex has been visited. These members are named `label` and `wasVisited`, respectively. The only function we need for the class is the constructor function that allows us to set the values for a vertex's data members. Here is the code for the `Vertex` class:

```
function Vertex(label) {
   this.label = label;
}
```

We will store the list of vertices in an array and will reference them in the `Graph` class by their position in the array.

Representing Edges

The real information about a graph is stored in the edges, since the edges describe the structure of a graph. As we mentioned earlier, it is tempting to represent a graph as a binary tree, but doing so is a mistake. A binary tree has a mostly fixed representation, since a parent node can have only two child nodes, while a graph structure provides much more flexibility. There can be many edges linked to a single vertex or just one edge, for example.

The method we will use for representing the edges of a graph is called an *adjacency list*, or an *array of adjacency lists*. With this method, the edges are stored as a vertex-indexed array of lists (arrays) of the vertices adjacent to each vertex. Using this scheme, when we reference a vertex in a program, we can efficiently access the list of all the vertices it is connected to. For example, if the vertex 2 is connected to vertices 0, 1, 3, and 4, and is stored in array position 2, accessing this element gives us access to an array stored at array position 2 that consists of the vertices 0, 1, 3, and 4. This is the representation method we choose to use in this chapter and is shown in Figure 11-3.

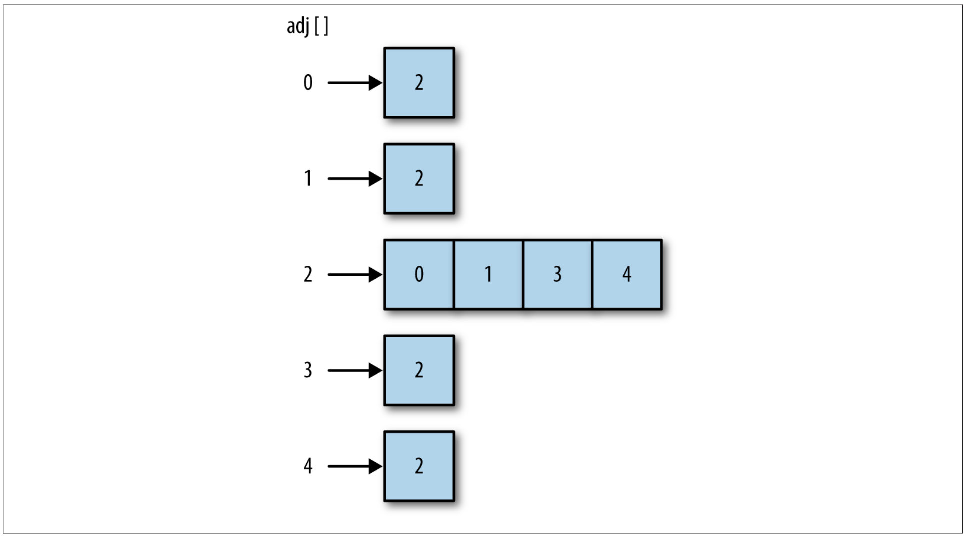

Figure 11-3. An adjacency list

Another method for representing the the edges of a graph is called an *adjacency matrix*. This is a two-dimensional array in which the elements of the array indicate whether an edge exists between two vertices.

Building a Graph

Once the decision is made on how to represent a graph in code, building a class to represent a graph is straightforward. Here is a first definition of a Graph class:

```
function Graph(v) {
   this.vertices = v;
   this.edges = 0;
   this.adj = [];
   for (var i = 0; i < this.vertices; ++i) {
      this.adj[i] = [];
      this.adj[i].push("");
   }
   this.addEdge = addEdge;
   this.toString = toString;
}
```

The class keeps track of how many edges are represented in a graph, as well as the number of vertices, by utilizing an array whose length is equal to the number of vertices in the graph. In each element of the array, the for loop adds a subarray to store all the adjacent vertices, and it initializes each element to the empty string.

The addEdge() function is defined as:

```
function addEdge(v,w) {
   this.adj[v].push(w);
   this.adj[w].push(v);
   this.edges++;
}
```

When this function is called with two vertices, A and B, the function finds the adjacency list for vertex A and adds B to the list, then it finds the adjacency list for B and adds A to the list. Finally, the function increments the number of edges by 1.

The showGraph() function displays the graph by showing a list of all vertices and the vertices that are adjacent to them:

```
function showGraph() {
   for (var i = 0; i < this.vertices; ++i) {
      putstr(i + " -> ");
      for (var j = 0; j < this.vertices; ++j) {
         if (this.adj[i][j] != undefined)
            putstr(this.adj[i][j] + ' ');
      }
      print();
   }
}
```

Example 11-1 displays the complete definition for the Graph class.

Example 11-1. The `Graph` class

```
function Graph(v) {
   this.vertices = v;
   this.edges = 0;
   this.adj = [];
   for (var i = 0; i < this.vertices; ++i) {
      this.adj[i] = [];
      this.adj[i].push("");
   }
   this.addEdge = addEdge;
   this.showGraph = showGraph;
}


function addEdge(v,w) {
   this.adj[v].push(w);
   this.adj[w].push(v);
   this.edges++;
}

function showGraph() {
   for (var i = 0; i < this.vertices; ++i) {
      putstr(i + " -> ");
      for (var j = 0; j < this.vertices; ++j) {
         if (this.adj[i][j] != undefined)
            putstr(this.adj[i][j] + ' ');
      }
      print();
   }
}
```

Here is a test program that demonstrates how to use the `Graph` class:

```
load("Graph.js");
g = new Graph(5);
g.addEdge(0,1);
g.addEdge(0,2);
g.addEdge(1,3);
g.addEdge(2,4);
g.showGraph();
```

The output from this program is:

```
0 ->  1 2
1 ->  0 3
2 ->  0 4
3 ->  1
4 ->  2
```

The output shows that vertex 0 has edges to vertices 1 and 2; vertex 1 has edges to vertices 0 and 3; vertex 2 has edges to vertices 0 and 4; vertex 3 has an edge to vertex 1; and vertex 4 has an edge to vertex 2. Of course, there is some redundancy in this display, as an edge between 0 and 1, for example, is the same as an edge between 1 and 0. For just display purposes this is fine, but we will need to modify this output when we start exploring the paths found in a graph.

Searching a Graph

Determining which vertices can be reached from a specified vertex is a common activity performed on graphs. We might want to know which roads lead from one town to other towns on the map, or which flights can take us from one airport to other airports.

These operations are performed on a graph using a search algorithm. There are two fundamental searches we can perform on a graph: the *depth-first* search and the *breadth-first* search. In this section we examine both algorithms.

Depth-First Search

Depth-first search involves following a path from the beginning vertex until it reaches the last vertex, then backtracking and following the next path until it reaches the last vertex, and so on until there are no paths left. Here we are not "searching" for a particular item, but instead searching to see what paths we can follow in a graph. Figure 11-4 illustrates how depth-first search works.

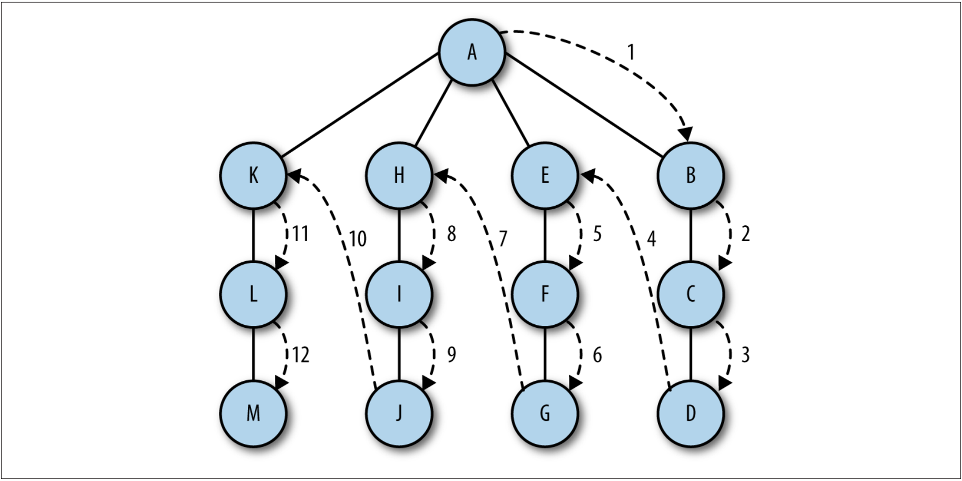

Figure 11-4. Depth-first search

The algorithm for performing a depth-first search is relatively simple—visit a vertex that has not already been visited, mark it as having been visited, then recursively visit the other unvisited vertices that are in the original vertex's adjacency list.

To make this algorithm work, we will need to add an array to our `Graph` class that stores visited vertices and initialize it to all `false` values. Here is a code fragment from the `Graph` class showing this new array and its initialization:

```
this.marked = [];
for (var i = 0; i < this.vertices; ++i) {
   this.marked[i] = false;
}
```

Now we can write the depth-first search function:

```
function dfs(v) {
   this.marked[v] = true;
   // if statement for print is not required
   if (this.adj[v] != undefined)
      print("Visited vertex: " + v);
   for each (var w in this.adj[v]) {
      if (!this.marked[w]) {
         this.dfs(w);
      }
   }
}
```

Notice that I've included a `print()` function so we can see the vertices as they're being visited. This function is, of course, not required for the `dfs()` function to work properly.

A program that demonstrates the `depthFirst()` function, along with the complete `Graph` class definition, is shown in Example 11-2.

Example 11-2. Performing a depth-first search

```
function Graph(v) {
   this.vertices = v;
   this.edges = 0;
   this.adj = [];
   for (var i = 0; i < this.vertices; ++i) {
      this.adj[i] = [];
      this.adj[i].push("");
   }
   this.addEdge = addEdge;
   this.showGraph = showGraph;
   this.dfs = dfs;
   this.marked = [];
   for (var i = 0; i < this.vertices; ++i) {
      this.marked[i] = false;
   }
}


function addEdge(v,w) {
   this.adj[v].push(w);
   this.adj[w].push(v);
   this.edges++;
}

function showGraph() {
   for (var i = 0; i < this.vertices; ++i) {
      putstr(i + " -> ");
      for (var j = 0; j

      < this.vertices; ++j) {
         if (this.adj[i][j] != undefined)
            putstr(this.adj[i][j] + ' ');
      }
      print();
   }
}

function dfs(v) {
   this.marked[v] = true;
   if (this.adj[v] != undefined) {
      print("Visited vertex: " + v);
   }
   for each (var w in this.adj[v]) {
      if (!this.marked[w]) {
         this.dfs(w);
      }
   }
}

// program to test dfs() function

load("Graph.js");
g = new Graph(5);
g.addEdge(0,1);
g.addEdge(0,2);
g.addEdge(1,3);
g.addEdge(2,4);
g.showGraph();
g.dfs(0);
```

The output from this program is:

```
0 ->  1 2
1 ->  0 3
2 ->  0 4
3 ->  1
4 ->  2
Visited vertex: 0
Visited vertex: 1
Visited vertex: 3
Visited vertex: 2
Visited vertex: 4
```

Breadth-First Search

A breadth-first search starts at a first vertex and tries to visit vertices as close to the first vertex as possible. In essence, this search moves through a graph layer by layer, first examining layers closer to the first vertex and then moving down to the layers farthest away from the starting vertex. Figure 11-5 demonstrates how breadth-first search works.

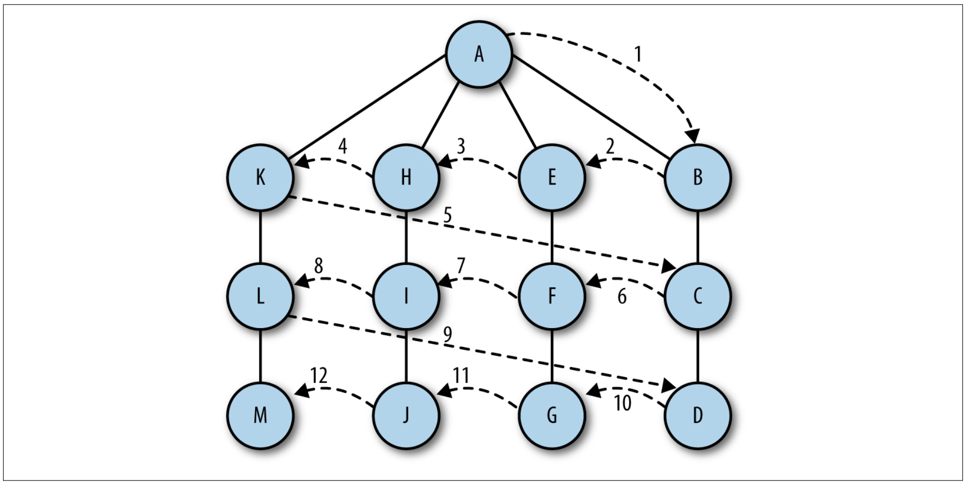

Figure 11-5. Breadth-first search

The algorithm for breadth-first search uses a queue abstraction instead of an array abstraction for storing visited vertices. The algorithm works as follows:

1. Find an unvisited vertex that is adjacent to the current vertex, add it to the list of visited vertices, and add it to the queue.
2. Take the next vertex, *v*, from the graph and add it to the list of visited vertices.
3. Add all unmarked vertices that are are adjacent to v and add them to the queue.

Here is the definition for the breadth-first search function:

```
function bfs(s) {
   var queue = [];
   this.marked[s] = true;
   queue.push(s); // add to back of queue
   while (queue.length > 0) {
      var v = queue.shift(); // remove from front of queue
      if (v == undefined) {
         print("Visited vertex: " + v);
      }
      for each (var w in this.adj[v]) {
         if (!this.marked[w]) {
                this.edgeTo[w] = v;
                this.marked[w] = true;
                queue.push(w);
            }
        }
    }
}
```

A test program for the breadth-first search function is shown in Example 11-3.

Example 11-3. Performing a breadth-first search

```
load("Graph.js");
g = new Graph(5);
g.addEdge(0,1);
g.addEdge(0,2);
g.addEdge(1,3);
g.addEdge(2,4);
g.showGraph();
g.bfs(0);
```

The output from this program is:

```
    0 ->  1 2
    1 ->  0 3
    2 ->  0 4
    3 ->  1
    4 ->  2
    Visited vertex: 0
    Visited vertex: 1
    Visited vertex: 2
    Visited vertex: 3
    Visited vertex: 4
```

Finding the Shortest Path

One of the most common operations performed on graphs is finding the shortest path from one vertex to another. Consider the following example: for vacation, you are going to travel to 10 major-league cities to watch baseball games over a two-week period. You want to minimize the number of miles you have to drive to visit all 10 cities using a shortest-path algorithm. Another shortest-path problem involves creating a network of computers, where the cost could be the time to transmit data between two computers or the cost of establishing and maintaining the connection. A shortest-path algorithm can determine the most effective way to build the network.

Breadth-First Search Leads to Shortest Paths

When we perform a breadth-first search, we are automatically finding the shortest paths from one vertex to another connected vertex. For example, when we want to find the shortest path from vertex A to vertex D, we first look for any one-edge paths from A to

D, then two-edge paths from A to D, and so on. This is exactly the way breadth-first search works, so we can easily modify the breadth-first search algorithm to find shortest paths.

Determining Paths

To find the shortest path, we need to modify the breadth-first search algorithm so that it records the paths that lead from one vertex to another vertex. This requires a few modifications to the `Graph` class.

First, we need an array that keeps track of edges from one vertex to the next. We'll name this array `edgeTo`. As we work through the breadth-first search function, every time we come across a vertex that is not marked, besides marking it, we will add an edge to that vertex from the vertex that we are exploring in the adjacency list. Here is the new `bfs()` function, along with the code you need to add to the `Graph` class:

```
// add this to Graph class
this.edgeTo = [];

// bfs function
function bfs(s) {
   var queue = [];
   this.marked[s] = true;
   queue.push(s); // add to back of queue
   while (queue.length > 0) {
      var v = queue.shift(); // remove from front of queue
      if (v == undefined) {
         print("Visited vertex: " + v);
      }
      for each (var w in this.adj[v]) {
         if (!this.marked[w]) {
            this.edgeTo[w] = v;
            this.marked[w] = true;
            queue.push(w);
         }
      }
   }
}
```

Now we need a function that can show us the paths that connect the different vertices of a graph. This function, `pathTo()`, creates a stack that stores all the vertices that have edges in common with a specified vertex. Here is the code for the function, along with a simple helper function:

```
function pathTo(v) {
   var source = 0;
   if (!this.hasPathTo(v)) {
      return undefined;
   }
   var path = [];
```

```
      for (var i = v; i != source; i = this.edgeTo[i]) {
         path.push(i);
      }
      path.push(s);
      return path;
   }

   function hasPathTo(v) {
      return this.marked[v];
   }
```

Be sure to add the appropriate declarations to the `Graph()` function:

```
   this.pathTo = pathTo;
   this.hasPathTo = hasPathTo;
```

With this function, all we have to do is write some client code to show the shortest path from the source to a particular vertex. Example 11-4 shows a program that creates a graph and shows the shortest path for a specified vertex.

Example 11-4. Finding the shortest path for a vertex

```
load("Graph.js");
g = new Graph(5);
g.addEdge(0,1);
g.addEdge(0,2);
g.addEdge(1,3);
g.addEdge(2,4);
var vertex = 4;
var paths = g.pathTo(vertex);
while (paths.length > 0) {
   if (paths.length > 1) {
      putstr(paths.pop() + '-');
   }
   else {
      putstr(paths.pop());
   }
}
```

The output from this program is:

```
   0-2-4
```

which is the shortest path from the source vertex 0 to vertex 4.

Topological Sorting

Topological sorting puts the vertices of a directed graph into an order such that all the directed edges point from a vertex earlier in the order to a vertex later in the order. For example, Figure 11-6 shows a directed-graph model of a typical computer science curriculum.

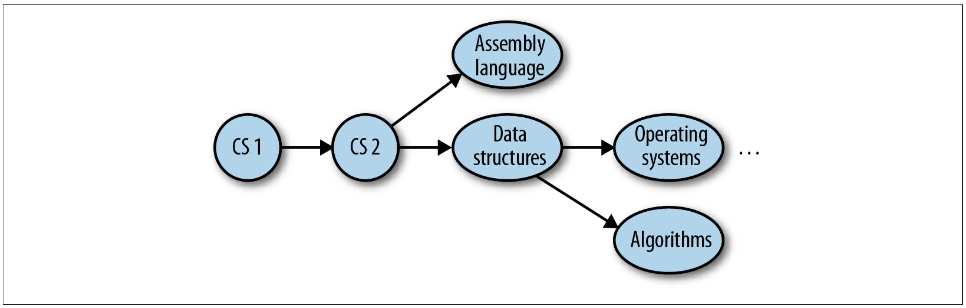

Figure 11-6. A directed graph model of a computer science curriculum

A topological sort of this graph would result in the following sequence:

1. CS 1
2. CS 2
3. Assembly language
4. Data structures
5. Operating systems
6. Algorithms

Courses 3 and 4 can be taken at the same time, as can courses 5 and 6.

This type of problem is called *precedence-constrained scheduling*, and every college student is familiar with it. You can't take English Composition II until you've taken English Composition I.

An Algorithm for Topological Sorting

The algorithm for topological sorting is similar to the algorithm for depth-first search. However, instead of immediately printing a vertex as it is visited, the algorithm visits all the adjacent vertices to the current vertex, and once that list is exhausted, we push the current vertex onto a stack.

Implementing the Topological Sorting Algorithm

The topological sort algorithm is broken up into two functions. The first function, `topSort()`, sets up the sorting process and calls a helper function, `topSortHelper()`, and then displays the sorted list of vertices.

The major work is done in the recursive function `topSortHelper()`. This function marks the current vertex as visited and then recursively visits each adjacent vertex in the current vertex's adjacency list, marking them as visited. Finally, the current vertex is pushed onto a stack.

Example 11-5 shows the code for the two functions.

Example 11-5. `topSort()` and `topSortHelper()`

```
function topSort() {
   var stack = [];
   var visited = [];
   for (var i = 0; i < this.vertices; i++) {
      visited[i] = false;
   }
   for (var i = 0; i < this.vertices; i++) {
      if (visited[i] == false) {
         this.topSortHelper(i, visited, stack);
      }
   }
   for (var i = 0; i < stack.length; i++) {
      if (stack[i] != undefined && stack[i] != false) {
         print(this.vertexList[stack[i]]);
      }
   }
}

function topSortHelper(v, visited, stack) {
   visited[v] = true;
   for each (var w in this.adj[v]) {
      if (!visited[w]) {
         this.topSortHelper(visited[w], visited, stack);
      }
   }
   stack.push(v);
}
```

The `Graph` class has also been modified so that we can work with symbolic vertices and not just numbers. Inside the code, each vertex is still only numbered, but we add an array, `vertexList`, which associates each vertex with a symbol (for our example it's a course name).

To make sure the new definition of the class is clear, we present the full definition, including the functions for topological sorting, below. The definition of the function `showGraph()` has changed so that symbolic names are shown instead of just vertex numbers. Example 11-6 shows the code.

Example 11-6. The `Graph` class

```
function Graph(v) {
   this.vertices = v;
   this.vertexList = [];
   this.edges = 0;
   this.adj = [];
   for (var i = 0; i < this.vertices; ++i) {
      this.adj[i] = [];
      this.adj[i].push("");
   }
   this.addEdge = addEdge;
   this.showGraph = showGraph;
   this.dfs = dfs;
   this.marked = [];
   for (var i = 0; i < this.vertices; ++i) {
      this.marked[i] = false;
   }
   this.bfs = bfs;
   this.edgeTo = [];
   this.hasPathTo = hasPathTo;
   this.pathTo = pathTo;
   this.topSortHelper = topSortHelper;
   this.topSort = topSort;

}

function topSort() {
   var stack = [];
   var visited = [];
   for (var i = 0; i < this.vertices; i++) {
      visited[i] = false;
   }
   for (var i = 0; i < this.vertices; i++) {
      if (visited[i] == false) {
         this.topSortHelper(i, visited, stack);
      }
   }
   for (var i = 0; i < stack.length; i++) {
      if (stack[i] != undefined && stack[i] != false) {
         print(this.vertexList[stack[i]]);
      }
   }
}

function topSortHelper(v, visited, stack) {
   visited[v] = true;
   for each (var w in this.adj[v]) {
      if (!visited[w]) {
         this.topSortHelper(visited[w], visited, stack);
      }
   }
   stack.push(v);
}

function addEdge(v,w) {
   this.adj[v].push(w);
   this.adj[w].push(v);
   this.edges++;
}

/*function showGraph() {
   for (var i = 0; i < this.vertices; ++i) {
      putstr(i + " -> ");
      for (var j = 0; j < this.vertices; ++j) {
         if (this.adj[i][j] != undefined)
            putstr(this.adj[i][j] + ' ');
      }
      print();
   }
}*/

// a new function to display symbolic names instead of numbers
function showGraph() {
   var visited = [];
   for (var i = 0; i < this.vertices; ++i) {
      putstr(this.vertexList[i] + " -> ");
      visited.push(this.vertexList[i]);
      for (var j = 0; j < this.vertices; ++j) {
         if (this.adj[i][j] != undefined) {
            if (visited.indexOf(this.vertexList[j]) < 0) {
               putstr(this.vertexList[j] + ' ');
            }
         }
      }
      print();
      visited.pop();
   }
}

function dfs(v) {
   this.marked[v] = true;
   if (this.adj[v] != undefined) {
      print("Visited vertex: " + v);
   }
   for each (var w in this.adj[v]) {
      if (!this.marked[w]) {
         this.dfs(w);
      }
   }
}

function bfs(s) {
   var queue = [];
   this.marked[s] = true;
   queue.unshift(s);
   while (queue.length > 0) {
      var v = queue.shift();
      if (typeof(v) != "string") {
         print("Visited vertex: " + v);
      }
      for each (var w in this.adj[v]) {
         if (!this.marked[w]) {
            this.edgeTo[w] = v;
            this.marked[w] = true;
            queue.unshift(w);
         }
      }
   }
}

function hasPathTo(v) {
   return this.marked[v];
}

function pathTo(v) {
   var source = 0;
   if (!this.hasPathTo(v)) {
      return undefined;
   }
   var path = [];
   for (var i = v; i != source; i = this.edgeTo[i]) {
      path.push(i);
   }
   path.push(s);
   return path;
}
```

A program that tests our implementation of topological sorting is shown in Example 11-7.

Example 11-7. Topological sorting

```
load("Graph.js");
g = new Graph(6);
g.addEdge(1,2);
g.addEdge(2,5);
g.addEdge(1,3);
g.addEdge(1,4);
g.addEdge(0,1);
g.vertexList = ["CS1", "CS2", "Data Structures",
                "Assembly Language", "Operating Systems",
                "Algorithms"];
g.showGraph();
g.topSort();
```

The output from this program is:

```
CS1
CS2
Data Structures
Assembly Language
```

```
Operating Systems
Algorithms
```

Exercises

1. Write a program that determines which type of graph search is faster—breadth-first or depth-first. Test your program with graphs of many different sizes.
2. Write a program that stores a graph in a file.
3. Write a program that reads a graph from a file.
4. Build a graph that models the map of the area where you live. Determine the shortest path from a starting vertex to the last vertex.
5. Perform a depth-first search and a breadth-first search of the graph created in example 4.