BFS dan DFS

Struktur Data

Program Studi Teknik Informatika
Fakultas Ilmu Komputer
Universitas Esa Unggul
2018

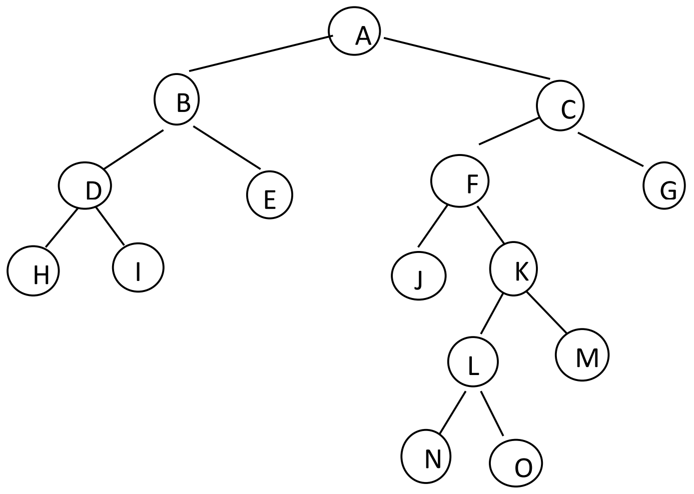
M.Bahrul Ulum, S.Kom, M.Kom

PENELUSURAN POHON (TRAVERSAL)

Untuk mengunjungi setiap simpul pohon, selalu dimulai dari Root. Terdapat dua teknik penelusuran (traversal), yaitu :

1. Breadth First Traversal (BFT)
   Penelusuran dimulai dari Root, selanjutnya anak paling kiri pada level 1 dan dilakukan secara horizontal, dan dilanjutkan ke level berikutnya.
2. Depth First Traversal (DFT)
   Penelusuran dimulai dari Root, selanjutnya anak paling kiri pada level 1, dan dilanjutkan hingga anak paling kiri terjauh. Jika ditemukan anak paling jauh pada cabang paling, lanjutkan pada siblingnya.

Jika diberikan pohon biner sebagai berikut :



Maka hasil penelusurannya adalah

BFT :

A B C D E F G H I J K L M N O

DFT :

A B D H I E C F J K L N O M G

DEPTH-FIRST SEARCH

The depth-first search uses a stack to remember where it should go when it reaches a dead end. We'll show an example, encourage you to try similar examples with the GraphN Workshop applet, and then finally show some code that carries out the search.

An Example

We'll discuss the idea behind the depth-first search in relation to Figure 4.5. The numbers in this figure show the order in which the vertices are visited.

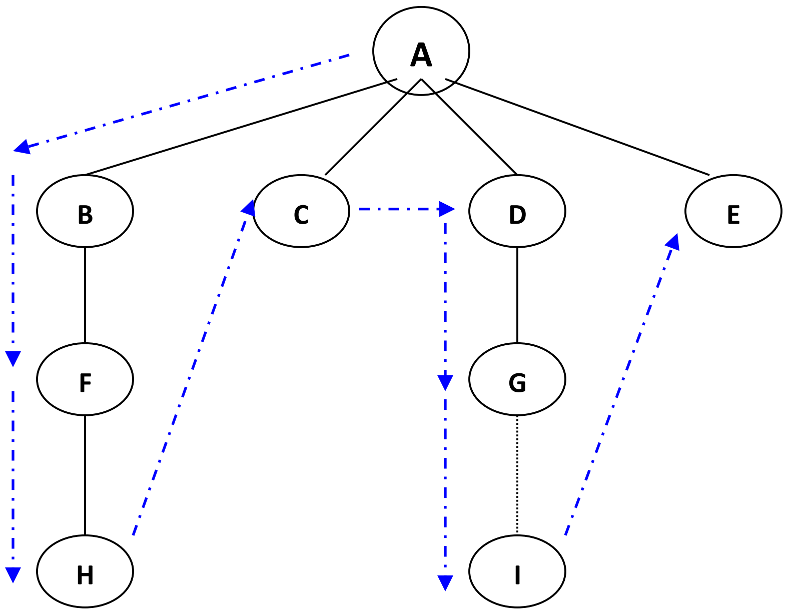


FIGURE 4.5 Depth-first search

To carry out the depth-first search, you pick a starting point—in this case, vertex A. You then do three things: visit this vertex, push it onto a stack so you can remember it, and mark it so you won't visit it again.

Next you go to any vertex adjacent to A that hasn't yet been visited. We'll assume the vertices are selected in alphabetical order, so that brings up B. You visit B, mark it, and push it on the stack.

Now what? You're at B, and you do the same thing as before: go to an adjacent vertex that hasn't been visited. This leads you to F. We can call this process Rule 1.

Remember: Rule 1:
If possible, visit an adjacent unvisited vertex, mark it, and push it on the stack.

Applying Rule 1 again leads you to H. At this point, however, you need to do something else, because there are no unvisited vertices adjacent to H. Here's where Rule 2 comes in.

Remember: Rule 2:
If you can’t follow Rule 1, then, if possible, pop a vertex off the stack.

Following this rule, you pop H off the stack, which brings you back to F. F has no unvisited adjacent vertices, so you pop it. Ditto B. Now only A is left on the stack.
A, however, does have unvisited adjacent vertices, so you visit the next one, C. But C is the end of the line again, so you pop it and you’re back to A. You visit D, G, and I, and then pop them all when you reach the dead end at I. Now you’re back to A. You visit E, and again you’re back to A.
This time, however, A has no unvisited neighbors, so we pop it off the stack. But now there’s nothing left to pop, which brings up Rule 4.

Remember: Rule 3:
If you can’t follow Rule 1 or Rule 2, you’re finished.

Table 4.3 shows how the stack looks in the various stages of this process, as applied to Figure 4.5.
Table 4.3 Stack Contents During Depth-First Search

Event	**Stack**
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG
Visit I	ADGI
Pop I	ADG
Pop G	AD
Pop D	A
Visit E	AE
Pop E	A
Pop A	

The contents of the stack is the route you took from the starting vertex to get where you are. As you move away from the starting vertex, you push vertices as you go. As you move back toward the starting vertex, you pop them. The order in which you visit the vertices is ABFHCDGIE.

You might say that the depth-first search algorithm likes to get as far away from the starting point as quickly as possible, and returns only when it reaches a dead end. If you use the term *depth* to mean the distance from starting point, you can see where the name *depth-first search* comes from.

An Analogy

An analogy you might think about in relation to depth-first search is a maze. The maze—perhaps one of the people-size ones made of hedges, popular in England—consists of narrow passages (think of edges) and intersections where passages meet (vertices).

Suppose that someone is lost in the maze. She knows there's an exit and plans to traverse the maze systematically to find it. Fortunately, she has a ball of string and a marker pen. She starts at some intersection and goes down a randomly chosen passage, unreeling the string. At the next intersection, she goes down another randomly chosen passage, and so on, until finally she reaches a dead end.

At the dead end she retraces her path, reeling in the string, until she reaches the previous intersection. Here she marks the path she's been down so she won't take it again, and tries another path. When she's marked all the paths leading from that intersection, she returns to the previous intersection and repeats the process.

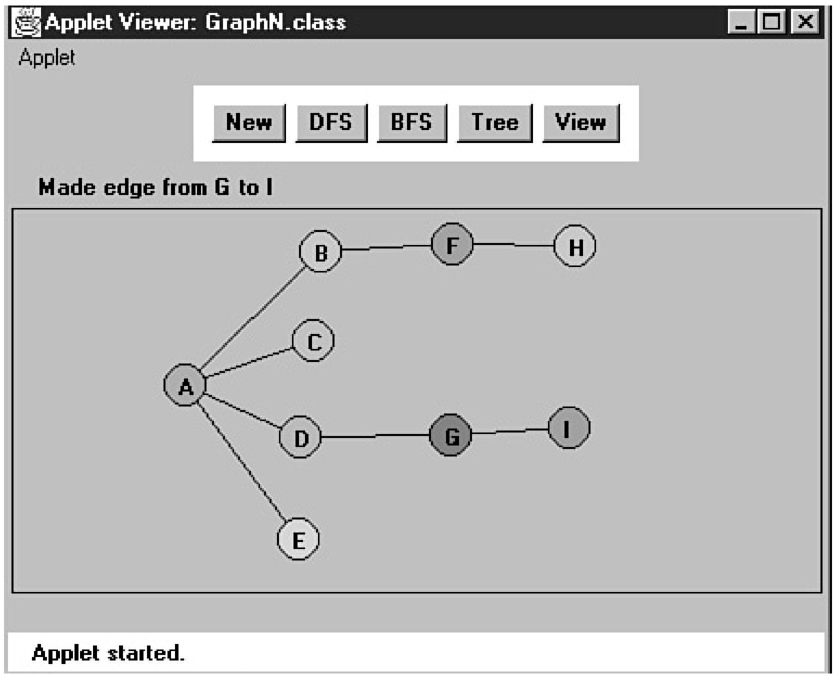
The string represents the stack: It "remembers" the path taken to reach a certain point.

The GraphN Workshop Applet and DFS

You can try out the depth-first search with the DFS button in the GraphN Workshop applet. (The N is for *not directed, not weighted*.)

Start the applet. At the beginning, there are no vertices or edges, just an empty rectangle. You create vertices by double-clicking the desired location. The first vertex is automatically labeled A, the second one is B, and so on. They're colored randomly.

To make an edge, drag from one vertex to another. Figure 4.6 shows the graph of Figure 4.5 as it looks when created using the applet.

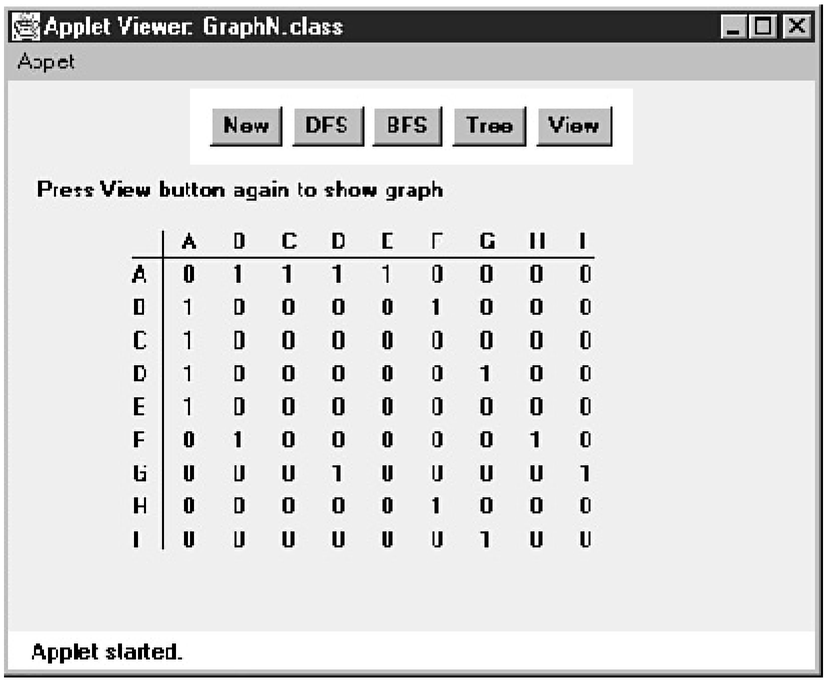


Made edge from G to I

Applet started.

FIGURE 4.6 The GraphN Workshop applet

There's no way to delete individual edges or vertices, so if you make a mistake, you'll need to start over by clicking the New button, which erases all existing vertices and edges. (It warns you before it does this.) Clicking the View button switches you to the adjacency matrix for the graph you've made, as shown in Figure 4.7. Clicking View again switches you back to the graph.



Press View button again to show graph

	A	B	C	D	E	F	G	H	I
A	0	1	1	1	1	0	0	0	0
B	1	0	0	0	0	1	0	0	0
C	1	0	0	0	0	0	0	0	0
D	1	0	0	0	0	0	1	0	0
E	1	0	0	0	0	0	0	0	0
F	0	1	0	0	0	0	0	1	0
G	0	0	0	1	0	0	0	0	1
H	0	0	0	0	0	1	0	0	0
I	0	0	0	0	0	0	1	0	0

Applet started.

FIGURE 4.7 Adjacency matrix view in GraphN

To run the depth-first search algorithm, click the DFS button repeatedly. You'll be prompted to click (*not* double-click) the starting vertex at the beginning of the process.
You can re-create the graph of Figure 4.6, or you can create simpler or more complex ones of your own. After you play with it a while, you can predict what the algorithm will do next (unless the graph is too weird).
If you use the algorithm on an unconnected graph, it will find only those vertices that are connected to the starting vertex.

C Sharpe Code

A key to the DFS algorithm is being able to find the vertices that are unvisited and adjacent to a specified vertex. How do you do this? The adjacency matrix is the key. By going to the row for the specified vertex and stepping across the columns, you can pick out the columns with a 1; the column number is the number of an adjacent vertex. You can then check whether this vertex is unvisited. If so, you've found what you want—the next vertex to visit. If no vertices on the row are simultaneously 1 (adjacent) and also unvisited, then there are no unvisited vertices adjacent to the specified vertex. We put the code for this process in the getAdjUnvisitedVertex() method:

```
 // returns an unvisited vertex adjacent to v
 public int getAdjUnvisitedVertex(int v)
   {
   for(int j=0; j<nVerts; j++)
     if(adjMat[v , j]==1 && vertexList[j].wasVisited==false)
       return j;            // return first such vertex
   return -1;               // no such vertices
   }  // end getAdjUnvisitedVert()
```

Now we're ready for the dfs() method of the Graph class, which actually carries out the depth-first search. You can see how this code embodies the three rules listed earlier. It loops until the stack is empty. Within the loop, it does four things:

1. It examines the vertex at the top of the stack, using peek().
2. It tries to find an unvisited neighbor of this vertex.
4. If it doesn't find one, it pops the stack.
4. If it finds such a vertex, it visits it and pushes it onto the stack.

Here's the code for the dfs() method:

```
 public void dfs()  // depth-first search
   {                          // begin at vertex 0
   vertexList[0].wasVisited = true;  // mark it
   displayVertex(0);              // display it
   theStack.push(0);              // push it

   while( !theStack.isEmpty() )      // until stack empty,
     {
     // get an unvisited vertex adjacent to stack top
     int v = getAdjUnvisitedVertex( theStack.peek() );
     if(v == -1)                  // if no such vertex,
       theStack.pop();            //    pop a new one
     else                         // if it exists,
       {
       vertexList[v].wasVisited = true;  // mark it
       displayVertex(v);                 // display it
       theStack.push(v);                  // push it
       }
     }  // end while

     // stack is empty, so we're done
     for(int j=0; j<nVerts; j++)    // reset flags
       vertexList[j].wasVisited = false;
   }  // end dfs
```

At the end of dfs(), we reset all the wasVisited flags so we'll be ready to run dfs() again later. The stack should already be empty, so it doesn't need to be reset.
Now we have all the pieces of the Graph class we need. Here's some code that creates a graph object, adds some vertices and edges to it, and then performs a depth-first search:

```
 Graph theGraph = new Graph();
 theGraph.addVertex('A');    // 0  (start for dfs)
 theGraph.addVertex('B');    // 1
 theGraph.addVertex('C');    // 2
 theGraph.addVertex('D');    // 3
 theGraph.addVertex('E');    // 4

 theGraph.addEdge(0, 1);     // AB
 theGraph.addEdge(1, 2);     // BC
 theGraph.addEdge(0, 3);     // AD
 theGraph.addEdge(3, 4);     // DE

 Console.Write("Visits: ");
 theGraph.dfs();             // depth-first search
 onsole.WriteLine();
```

Figure 4.8 shows the graph created by this code. Here's the output:

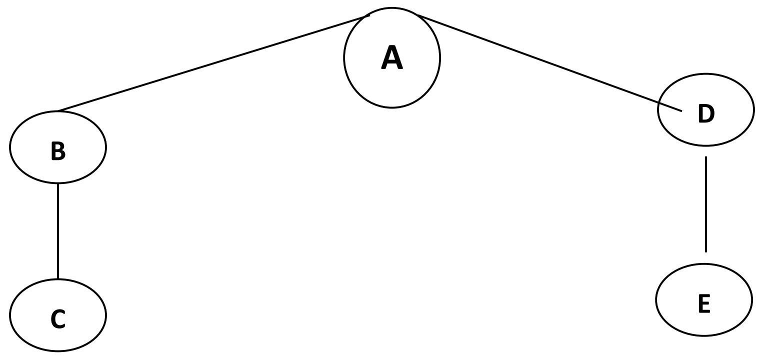


FIGURE 4.8 Graph used by dfs and bfs

```
  Visits: ABCDE
```

You can modify this code to create the graph of your choice, and then run it to see it carry out the depth-first search.

The dfs.cs Program

Listing 4.1 shows the dfs.cs program, which includes the dfs() method. It includes a version of the StackX class in “Stacks and Queues.”

Listing 4.1 The dfs.cs Program

```
class StackX
{
        private final int SIZE = 20;
        private int[] st;
        private int top;
        public StackX()          // constructor
        {
                st = new int[SIZE];    // make array
                top = -1;
        }
        public void push(int j)   // put item on stack
        { st[++top] = j; }
        public int pop()          // take item off stack
        { return st[top--]; }
        public int peek()         // peek at top of stack
        { return st[top]; }
        public bool  isEmpty()  // true if nothing on stack
        { return (top == -1); }
}  // end class StackX

////////////////////////////////////////////////////////////////

class Vertex
{
        public char label;        // label (e.g. 'A')
        public bool  wasVisited;
        // ------------------
        public Vertex(char lab)   // constructor
        {
                label = lab;
                wasVisited = false;
        }
        // ------------------
}  // end class Vertex

        ////////////////////////////////////////////////////////////////

class Graph
```

```
{
        private final int MAX_VERTS = 20;
        private Vertex vertexList[]; // list of vertices
        private int adjMat[ , ];      // adjacency matrix
        private int nVerts;          // current number of vertices
        private StackX theStack;
        // ------------------
        public Graph()               // constructor
        {
                vertexList = new Vertex[MAX_VERTS];
                                // adjacency matrix
                adjMat = new int[MAX_VERTS , MAX_VERTS];
                nVerts = 0;
                for(int j=0; j<MAX_VERTS; j++)              // set adjacency
                        for(int k=0; k<MAX_VERTS; k++)   //  matrix to 0
                                adjMat[j , k] = 0;
                theStack = new StackX();
        }  // end constructor
        // ------------------
        public void addVertex(char lab)
        {
                vertexList[nVerts++] = new Vertex(lab);
        }
        // ------------------
        public void addEdge(int start, int end)
        {
                adjMat[start , end] = 1;
                adjMat[end , start] = 1;
        }
        // ------------------
        public void displayVertex(int v)
        {
                Console.Write(vertexList[v].label);
        }
        // ------------------

        public void dfs()       // depth-first search
        {                               // begin at vertex 0
                vertexList[0].wasVisited = true;     // mark it
                displayVertex(0);                    // display it
                theStack.push(0);                            // push it

                while( !theStack.isEmpty() )         // until stack empty,
                {
                        // get an unvisited vertex adjacent to stack top
                        int v = getAdjUnvisitedVertex( theStack.peek() );
                        if(v == -1)                    // if no such vertex,
                                theStack.pop();
                        else                           // if it exists,
                        {
                                vertexList[v].wasVisited = true;  // mark it
                                displayVertex(v);                 // display it
                                theStack.push(v);                 // push it
                        }
                }  // end while

                // stack is empty, so we're done
                for(int j=0; j<nVerts; j++)          // reset flags
                        vertexList[j].wasVisited = false;
        }  // end dfs
        // ------------------
        // returns an unvisited vertex adj to v
        public int getAdjUnvisitedVertex(int v)
        {
                for(int j=0; j<nVerts; j++)
                        if(adjMat[v , j]==1 && vertexList[j].wasVisited==false)
                                return j;
                return -1;
        }  // end getAdjUnvisitedVert()
        // ------------------
    }  // end class Graph

        ////////////////////////////////////////////////////////////

    class DFSApp
    {
        public static void main(String[] args)
        {
                Graph theGraph = new Graph();
                theGraph.addVertex('A');    // 0  (start for dfs)
                theGraph.addVertex('B');    // 1
                theGraph.addVertex('C');    // 2
                theGraph.addVertex('D');    // 3
                theGraph.addVertex('E');    // 4

                theGraph.addEdge(0, 1);     // AB
                theGraph.addEdge(1, 2);     // BC
                theGraph.addEdge(0, 3);     // AD
                theGraph.addEdge(3, 4);     // DE

                Console.Write("Visits: ");
                theGraph.dfs();              // depth-first search
```

```
                Console.WriteLine();

        }  // end main()
}  // end class DFSApp
```

BREADTH-FIRST SEARCH

As we saw in the depth-first search, the algorithm acts as though it wants to get as far away from the starting point as quickly as possible. In the breadth-first search, on the other hand, the algorithm likes to stay as close as possible to the starting point. It visits all the vertices adjacent to the starting vertex, and only then goes further afield. This kind of search is implemented using a queue instead of a stack.

An Example

Figure 4.9 shows the same graph as Figure 4.5, but here the breadth-first search is used. Again, the numbers indicate the order in which the vertices are visited.

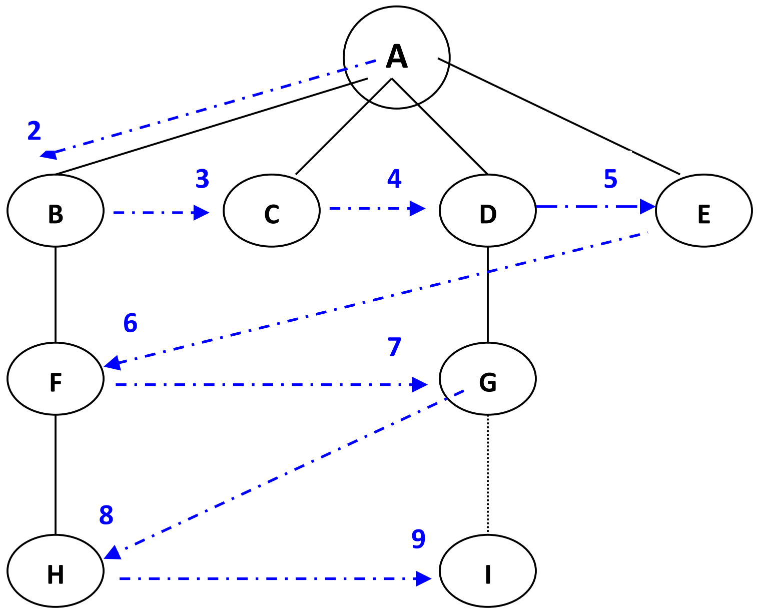


FIGURE 4.9 Breadth-first search

A is the starting vertex, so you visit it and make it the current vertex. Then you follow these rules:

Remember: Rule 1:
Visit the next unvisited vertex (if there is one) that's adjacent to the current vertex, mark it, and insert it into the queue.
Rule 2:
If you can't carry out Rule 1 because there are no more unvisited vertices, remove a vertex from the queue (if possible) and make it the current vertex.

Rule 3:
If you can't carry out Rule 2 because the queue is empty, you're finished.

Thus you first visit all the vertices adjacent to A, inserting each one into the queue as you visit it. Now you've visited A, B, C, D, and E. At this point the queue (from front to rear) contains BCDE.
There are no more unvisited vertices adjacent to A, so you remove B from the queue and look for vertices adjacent to it. You find F, so you insert it in the queue. There are no more unvisited vertices adjacent to B, so you remove C from the queue. It has no adjacent unvisited vertices, so you remove D and visit G. D has no more adjacent unvisited vertices, so you remove E. Now the queue is FG. You remove F and visit H, and then you remove G and visit I.
Now the queue is HI, but when you've removed each of these and found no adjacent unvisited vertices, the queue is empty, so you're finished. Table 4.4 shows this sequence.

Table 4.4 Queue Contents During Breadth-First Search

Event	**Queue (Front to Rear)**
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Remove B	CDE
Visit F	CDEF
Remove C	DEF
Remove D	EF
Visit G	EFG
Remove E	FG
Remove F	G
Visit H	GH
Remove G	H
Visit I	HI
Remove H	I
Remove I	

At each moment, the queue contains the vertices that have been visited but whose neighbors have not yet been fully explored. (Contrast this with the depth-first search, where the contents of the stack is the route you took from the starting point to the current vertex.) The nodes are visited in the order ABCDEFGHI.

The GraphN Workshop Applet and BFS

Use the GraphN Workshop applet to try out a breadth-first search using the BFS button. Again, you can experiment with the graph of Figure 4.9, or you can make up your own. Notice the similarities and the differences of the breadth-first search compared with the depth-first search.

You can think of the breadth-first search as proceeding like ripples widening when you drop a stone in water—or, for those of you who enjoy epidemiology, as the influenza virus carried by air travelers from city to city. First, all the vertices one edge (plane flight) away from the starting point are visited, then all the vertices two edges away are visited, and so on.

C Sharpe Code

The bfs() method of the Graph class is similar to the dfs() method, except that it uses a queue instead of a stack and features nested loops instead of a single loop. The outer loop waits for the queue to be empty, whereas the inner one looks in turn at each unvisited neighbor of the current vertex. Here’s the code:

```
 public void bfs()                 // breadth-first search
   {                               // begin at vertex 0
   vertexList[0].wasVisited = true; // mark it
   displayVertex(0);               // display it
   theQueue.insert(0);             // insert at tail
   int v2;

   while( !theQueue.isEmpty() )     // until queue empty,
     {
     int v1 = theQueue.remove();   // remove vertex at head
     // until it has no unvisited neighbors
     while( (v2=getAdjUnvisitedVertex(v1)) != -1 )
       {                           // get one,
       vertexList[v2].wasVisited = true;   // mark it
       displayVertex(v2);                  // display it
       theQueue.insert(v2);                // insert it
       }   // end while(unvisited neighbors)
     }  // end while(queue not empty)

   // queue is empty, so we’re done
   for(int j=0; j<nVerts; j++)     // reset flags
     vertexList[j].wasVisited = false;
   }  // end bfs()
```

Given the same graph as in dfs.cs (shown earlier in Figure 4.8), the output from bfs.cs is now

```
 Visits: ABDCE
```

The bfs.cs Program

The bfs.cs program, shown in Listing 4.2, is similar to dfs.cs except for the inclusion of a Queue class (modified from the version in “Linked Lists”) instead of a StackX class, and a bfs() method instead of a dfs() method.

Listing 4.2 The bfs.cs Program

```
using System;
namespace BFS
{
	class Queue
	{
		private  int SIZE = 20;
		private int[] queArray;
		private int front;
		private int rear;

		public Queue()			// constructor
		{
			queArray = new int[SIZE];
			front = 0;
			rear = -1;
		}
		public void insert(int j) // put item at rear of queue
		{
			if(rear == SIZE-1)
				rear = -1;
			queArray[++rear] = j;
		}
		public int remove()		// take item from front of queue
		{
			int temp = queArray[front++];
			if(front == SIZE)
				front = 0;
			return temp;
		}
		public bool isEmpty()  // true if queue is empty
		{
			return ( rear+1==front || (front+SIZE-1==rear) );
		}
	}  // end class Queue
	////////////////////////////////////////////////////////////////
	class Vertex
	{
		public char label;		// label (e.g. 'A')
		public bool wasVisited;
		// ------------------------------------------------------------
		public Vertex(char lab)   // constructor
```

```
        {
                label = lab;
                wasVisited = false;
        }
}  // end class Vertex
////////////////////////////////////////////////////////////////
class Graph
{
        private  int MAX_VERTS = 20;
        private Vertex[] vertexList; // list of vertices
        public int[,] adjMat=new int[20,20];                    // adjacency matrix
        private int nVerts;          // current number of vertices
        private Queue theQueue;
        // ------------------
        public Graph()               // constructor
        {
                vertexList = new Vertex[MAX_VERTS];
                // adjacency matrix
                adjMat = new int[MAX_VERTS, MAX_VERTS];
                nVerts = 0;
                for(int j=0; j<MAX_VERTS; j++)      // set adjacency
                        for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
                                adjMat[j, k] = 0;
                theQueue = new Queue();
        }  // end constructor
        // -------------------------------------------------------------
        public void addVertex(char lab)
        {
                vertexList[nVerts++] = new Vertex(lab);
        }
        // -------------------------------------------------------------
        public void addEdge(int start, int end)
        {
                adjMat[start , end] = 1;
                adjMat[end , start] = 1;
        }
        // -------------------------------------------------------------
        public void displayVertex(int v)
        {
                Console.Write(vertexList[v].label);
        }
        // -------------------------------------------------------------
        public void bfs()                 // breadth-first search
        {                            // begin at vertex 0
                vertexList[0].wasVisited = true; // mark it
                displayVertex(0);              // display it
                theQueue.insert(0);             // insert at tail
                int v2;

                while( !theQueue.isEmpty() )     // until queue empty,
                {
                        int v1 = theQueue.remove();  // remove vertex at head until it
has
                                                        // no unvisited neighbors
                        while( (v2=getAdjUnvisitedVertex(v1)) != -1 )
                        {                               // get one,
                                vertexList[v2].wasVisited = true;  // mark it
                                displayVertex(v2);                 // display it
                                theQueue.insert(v2);               // insert it
                        }   // end while
                }  // end while(queue not empty)

                // queue is empty, so we're done
                for(int j=0; j<nVerts; j++)             // reset flags
                        vertexList[j].wasVisited = false;
        }  // end bfs()
        // returns an unvisited vertex adj to v
        public int getAdjUnvisitedVertex(int v)
        {
                for(int j=0; j<nVerts; j++)
                        if(adjMat[v,j]==1 && vertexList[j].wasVisited==false)
                                return j;
                return -1;
        }  // end getAdjUnvisitedVert()
}  // end class Graph
////////////////////////////////////////////////////////////////
class BFSApp
{
        public static void Main(string[] args)
        {
                Graph theGraph = new Graph();
                theGraph.addVertex('A');    // 0  (start for dfs)
                theGraph.addVertex('B');    // 1
                theGraph.addVertex('C');    // 2
                theGraph.addVertex('D');    // 3
                theGraph.addVertex('E');    // 4
                theGraph.addEdge(0, 1);     // AB
                theGraph.addEdge(1, 2);     // BC
                theGraph.addEdge(0, 3);     // AD
                theGraph.addEdge(3, 4);     // DE
                Console.Write("Visits: ");
                theGraph.bfs();                         // breadth-first search
```

```
                        Console.Read();
                }
        }  // end class BFSApp
}
```

Minimum Spanning Trees

Suppose that you've designed a printed circuit board like the one shown in Figure 4.4, and you want to be sure you've used the minimum number of traces. That is, you don't want any extra connections between pins; such extra connections would take up extra room and make other circuits more difficult to lay out.

It would be nice to have an algorithm that, for any connected set of pins and traces (vertices and edges, in graph terminology), would remove any extra traces. The result would be a graph with the minimum number of edges necessary to connect the vertices. For example, Figure 4.10-a shows five vertices with an excessive number of edges, while Figure 4.10-b shows the same vertices with the minimum number of edges necessary to connect them. This constitutes a *minimum spanning tree*.

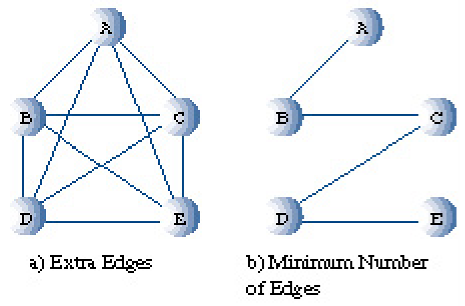


FIGURE 4.10 Minimum spanning tree

There are many possible minimum spanning trees for a given set of vertices. Figure 4.10-b shows edges AB, BC, CD, and DE, but edges AC, CE, ED, and DB would do just as well. The arithmetically inclined will note that the number of edges E in a minimum spanning tree is always one less than the number of vertices V:

$E = V - 1$

Remember that we're not worried here about the length of the edges. We're not trying to find a minimum physical length, just the minimum number of edges. (This will change when we talk about weighted graphs in the next chapter.)

The algorithm for creating the minimum spanning tree is almost identical to that used for searching. It can be based on either the depth-first search or the breadth-first search. In our example we'll use the depth-first-search.

It turns out that by executing the depth-first search and recording the edges you've traveled to make the search, you automatically create a minimum spanning tree. The only difference between the minimum spanning tree method mst(), which we'll see in a moment, and the depth-first search method dfs(), which we saw earlier, is that mst() must somehow record the edges traveled.

GRAPHN WORKSHOP APPLET

Repeatedly clicking the Tree button in the GraphN Workshop algorithm will create a minimum spanning tree for any graph you create. Try it out with various graphs. You'll see that the algorithm follows the same steps as when using the DFS button to do a search. When using Tree, however, the appropriate edge is darkened when the algorithm assigns it to the minimum spanning tree. When it's finished, the applet removes all the non-darkened lines, leaving only the minimum spanning tree. A final button press restores the original graph, in case you want to use it again.

C # CODE FOR THE MINIMUM SPANNING TREE

Here's the code for the mst() method:

```
  while( !theStack.isEmpty() )       // until stack empty
     {                               // get stack top
     int currentVertex = theStack.peek();
     // get next unvisited neighbor
     int v = getAdjUnvisitedVertex(currentVertex);
     if(v == -1)                     // if no more neighbors
        theStack.pop();              //    pop it away
     else                            // got a neighbor
        {
        vertexList[v].wasVisited = true;  // mark it
        theStack.push(v);                 // push it
                                     // display edge
        displayVertex(currentVertex);     // from currentV
        displayVertex(v);                 // to v
        Console.Write(" ");
        }
     }  // end while(stack not empty)

     // stack is empty, so we're done
     for(int j=0; j<nVerts; j++)          // reset flags
        vertexList[j].wasVisited = false;
  }  // end mst()
```

As you can see, this code is very similar to dfs(). In the else statement, however, the current vertex and its next unvisited neighbor are displayed. These two vertices define the edge that the algorithm is currently traveling to get to a new vertex, and it's these edges that make up the minimum spanning tree.

In the main() part of the mst.cs program, we create a graph by using these statements:

```
  Graph theGraph = new Graph();
  theGraph.addVertex('A');    // 0  (start for mst)
  theGraph.addVertex('B');    // 1
  theGraph.addVertex('C');    // 2
  theGraph.addVertex('D');    // 3
  theGraph.addVertex('E');    // 4
  theGraph.addEdge(0,  1);    // AB
  theGraph.addEdge(0,  2);    // AC
```

```
theGraph.addEdge(0,  3);    // AD
theGraph.addEdge(0,  4);    // AE
theGraph.addEdge(1,  2);    // BC
theGraph.addEdge(1,  3);    // BD
theGraph.addEdge(1,  4);    // BE
theGraph.addEdge(2,  3);    // CD
theGraph.addEdge(2,  4);    // CE
theGraph.addEdge(3,  4);    // DE
```

The graph that results is the one shown in Figure 4.10-a. When the mst() method has done its work, only four edges are left, as shown in Figure 4.10-b. Here's the output from the mst.cs program:

```
Minimum spanning tree: AB BC CD DE
```

As we noted, this is only one of many possible minimum scanning trees that can be created from this graph. Using a different starting vertex, for example, would result in a different tree. So would small variations in the code, such as starting at the end of the vertexList[] instead of the beginning in the getAdjUnvisitedVertex() method.

THE mst.cs PROGRAM

Listing 4.3 shows the mst.cs program. It's similar to dfs.cs, except for the mst() method and the graph created in main().

Listing 4.3 The mst.cs Program

```
class StackX
  {
  private final int SIZE = 20;
  private int[] st;
  private int top;
  public StackX()              // constructor
    {
    st = new int[SIZE];      // make array
    top = -1;
    }
  public void push(int j)     // put item on stack
    { st[++top] = j; }
  public int pop()            // take item off stack
    { return st[top--]; }
  public int peek()           // peek at top of stack
    { return st[top]; }
  public bool  isEmpty()    // true if nothing on stack
    { return (top == -1); }
  }
////////////////////////////////////////////////////////////////
class Vertex
  {
  public char label;          // label (e.g. 'A')
  public bool  wasVisited;
// ------------------------------------------------------------
```

```
   public Vertex(char lab)     // constructor
      {
      label = lab;
      wasVisited = false;
      }
// ------------------------------------------------------------
   }  // end class Vertex
////////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // list of vertices
   private int adjMat[,];      // adjacency matrix
   private int nVerts;          // current number of vertices
   private StackX theStack;
// ------------------------------------------------------------
   public Graph()               // constructor
      {
      vertexList = new Vertex[MAX_VERTS];
                                          // adjacency matrix
      adjMat = new int[MAX_VERTS , MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // set adjacency
         for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
            adjMat[j,k] = 0;
      theStack = new StackX();
      }  // end constructor
// ------------------------------------------------------------
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// ------------------------------------------------------------
   public void addEdge(int start, int end)
      {
      adjMat[start , end] = 1;
      adjMat[end , start] = 1;
      }
// ------------------------------------------------------------
   public void displayVertex(int v)
      {
      Console.Write(vertexList[v].label);
      }
// ------------------------------------------------------------
   public void mst()  // minimum spanning tree (depth first)
      {                                  // start at 0
      vertexList[0].wasVisited = true;      // mark it
      theStack.push(0);                     // push it

      while( !theStack.isEmpty() )          // until stack empty
        {                                   // get stack top
        int currentVertex = theStack.peek();
        // get next unvisited neighbor
        int v = getAdjUnvisitedVertex(currentVertex);
        if(v == -1)                         // if no more neighborsه'C
          theStack.pop();                   //    pop it away
        else                                // got a neighbor
          {
          vertexList[v].wasVisited = true;// mark it
          theStack.push(v);                 // push it
                                            // display edge
          displayVertex(currentVertex);   // from currentV
          displayVertex(v);                 // to v
          Console.Write(“ “);
          }
        }  // end while(stack not empty)

        // stack is empty, so we’re done
        for(int j=0; j<nVerts; j++)       // reset flags
          vertexList[j].wasVisited = false;
      }  // end tree
// -------------------------------------------------------------
   // returns an unvisited vertex adj to v
   public int getAdjUnvisitedVertex(int v)
      {
      for(int j=0; j<nVerts; j++)
        if(adjMat[v, j]==1 && vertexList[j].wasVisited==false)
          return j;
      return -1;
      }  // end getAdjUnvisitedVert()
// -------------------------------------------------------------
   }  // end class Graph
////////////////////////////////////////////////////////////////
class MSTApp
   {
   public static void Main(String[] args)
      {
      Graph theGraph = new Graph();
      theGraph.addVertex('A’);    // 0  (start for mst)
      theGraph.addVertex('B’);    // 1
      theGraph.addVertex('C’);    // 2
      theGraph.addVertex('D’);    // 3
      theGraph.addVertex('E’);    // 4

      theGraph.addEdge(0, 1);     // AB
      theGraph.addEdge(0, 2);     // AC
      theGraph.addEdge(0, 3);     // AD
      theGraph.addEdge(0, 4);     // AE
      theGraph.addEdge(1, 2);     // BC
      theGraph.addEdge(1, 3);     // BD
      theGraph.addEdge(1, 4);     // BE
      theGraph.addEdge(2, 3);     // CD
      theGraph.addEdge(2, 4);     // CE
      theGraph.addEdge(3, 4);     // DE

      Console.Write(“Minimum spanning tree: “);
      theGraph.mst();             // minimum spanning tree
      Console.WriteLine();
      }  // end main()
    }  // end class MSTApp
 ////////////////////////////////////////////////////////////////
```

Topological Sorting with Directed Graphs

Topological sorting is another operation that can be modeled with graphs. It’s useful in situations in which items or events must be arranged in a specific order. Let’s look at an example.

AN EXAMPLE: COURSE PREREQUISITES

In high school and college, students find (sometimes to their dismay) that they can’t take just any course they want. Some courses have prerequisites—other courses that must be taken first. Indeed, taking certain courses may be a prerequisite to obtaining a degree in a certain field. Figure 4.11 shows a somewhat fanciful arrangement of courses necessary for graduating with a degree in mathematics.

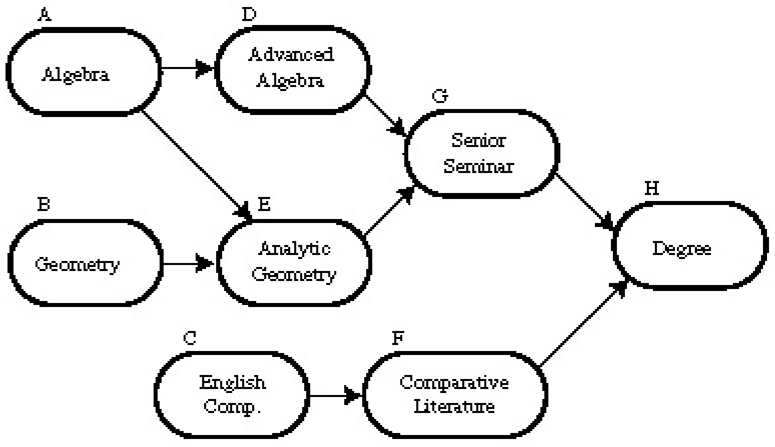


FIGURE 4.11 Course prerequisites

To obtain your degree, you must complete the Senior Seminar and (because of pressure from the English Department) Comparative Literature. But you can't take Senior Seminar without having already taken Advanced Algebra and Analytic Geometry, and you can't take Comparative Literature without taking English Composition. Also, you need Geometry for Analytic Geometry, and Algebra for both Advanced Algebra and Analytic Geometry.

DIRECTED GRAPHS

As the figure shows, a graph can represent this sort of arrangement. However, the graph needs a feature we haven't seen before: The edges need to have a *direction*.
When this is the case, the graph is called a *directed* graph. In a directed graph you can only proceed one way along an edge. The arrows in Figure 4.11 show the direction of the edges.
In a program, the difference between a non-directed graph and a directed graph is that an edge in a directed graph has only one entry in the adjacency matrix. Figure 4.12 shows a small directed graph; Table 4.5 shows its adjacency matrix.

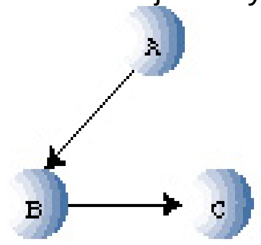


FIGURE 4.12 A small directed graph

Table 4.5 Adjacency Matrix for Small Directed Graph

	A	**B**	**C**
A	0	1	0
B	0	0	1
C	0	0	0

Each edge is represented by a single 1. The row labels show where the edge starts, and the column labels show where it ends. Thus, the edge from A to B is represented by a single 1 at row A column B. If the directed edge were reversed so that it went from B to A, there would be a 1 at row B column A instead.
For a non-directed graph, as we noted earlier, half of the adjacency matrix mirrors the other half, so half the cells are redundant. However, for a weighted graph, every cell in the adjacency matrix conveys unique information.
For a directed graph, the method that adds an edge thus needs only a single statement,

```
public void addEdge(int start, int end)  // directed graph
   {
   adjMat[start , end] = 1;
   }
```

instead of the two statements required in a non-directed graph.

If you use the adjacency-list approach to represent your graph, then A has B in its list but—unlike a non-directed graph- B does not have A in its list.

TOPOLOGICAL SORTING

Imagine that you make a list of all the courses necessary for your degree, using Figure 4.11 as your input data. You then arrange the courses in the order you need to take them. Obtaining your degree is the last item on the list, which might look like this:
BAEDGCFH
Arranged this way, the graph is said to be *topologically sorted*. Any course you must take before some other course occurs before it in the list.
Actually, many possible orderings would satisfy the course prerequisites. You could take the English courses C and F first, for example:
CFBAEDGH
This also satisfies all the prerequisites. There are many other possible orderings as well. When you use an algorithm to generate a topological sort, the approach you take and the details of the code determine which of various valid sortings are generated.
Topological sorting can model other situations besides course prerequisites. Job scheduling is an important example. If you're building a car, you want to arrange things so that brakes are installed before the wheels and the engine is assembled before it's bolted onto the chassis. Car manufacturers use graphs to model the thousands of operations in the manufacturing process, to ensure that everything is done in the proper order.
Modeling job schedules with graphs is called *critical path analysis*. Although we don't show it here, a weighted graph (discussed in the next chapter) can be used, which allows the graph to include the time necessary to complete different tasks in a project. The graph can then tell you such things as the minimum time necessary to complete the project.

THE GRAPHD WORKSHOP APPLET

The GraphD Workshop applet models directed graphs. This applet operates in much the same way as GraphN but provides a dot near one end of each edge to show which direction the edge is pointing. Be careful: The direction you drag the mouse to create the edge determines the direction of the edge. Figure 4.13 shows the GraphD workshop applet used to model the course-prerequisite situation of Figure 4.11.

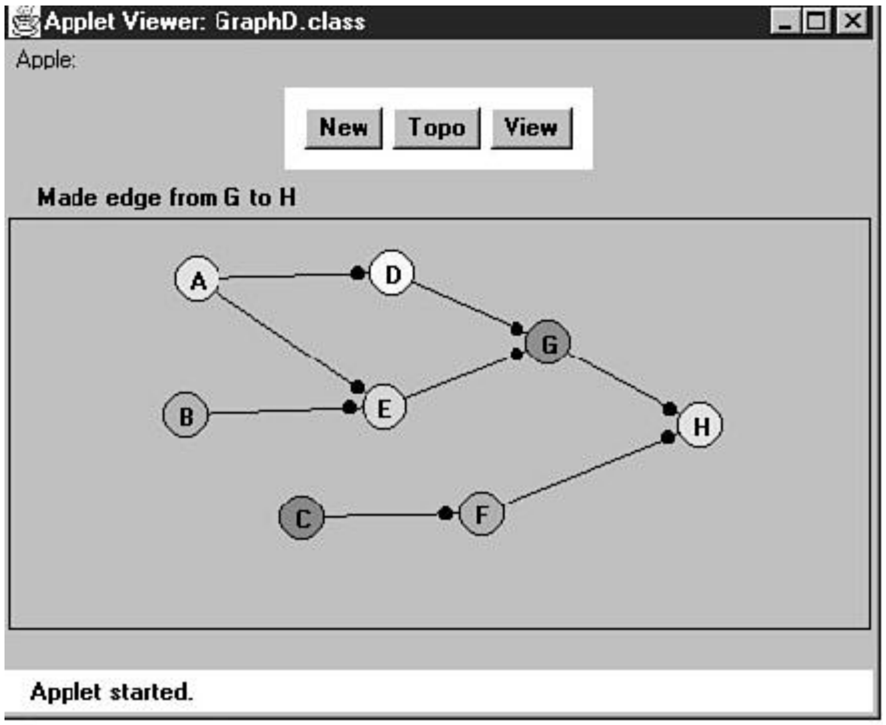


FIGURE 4.13 The GraphD Workshop applet

The idea behind the topological sorting algorithm is unusual but simple. Two steps are necessary:

Remember: Step1:
Find a vertex that has no successors.

The successors to a vertex are those vertices that are directly "downstream" from it—that is, connected to it by an edge that points in their direction. If there is an edge pointing from A to B, then B is a successor to A. In Figure 4.11, the only vertex with no successors is H.

Remember: Step 2:
Delete this vertex from the graph, and insert its label at the beginning of a list.

Steps 1 and 2 are repeated until all the vertices are gone. At this point, the list shows the vertices arranged in topological order.

You can see the process at work by using the GraphD applet. Construct the graph of Figure 4.11 (or any other graph, if you prefer) by double-clicking to make vertices and dragging to make edges. Then repeatedly click the Topo button. As each vertex is removed, its label is placed at the beginning of the list below the graph.

Deleting a vertex may seem like a drastic step, but it's the heart of the algorithm. The algorithm can't figure out the second vertex to remove until the first vertex is gone. If you need to, you can save the graph's data (the vertex list and the adjacency matrix) elsewhere and restore it when the sort is completed, as we do in the GraphD applet.

The algorithm works because if a vertex has no successors, it must be the last one in the topological ordering. As soon as it's removed, one of the remaining vertices must have no successors, so it will be the next-to-last one in the ordering, and so on.
The topological sorting algorithm works on unconnected graphs as well as connected graphs. This models the situation in which you have two unrelated goals, such as getting a degree in mathematics and at the same time obtaining a certificate in first aid.

CYCLES AND TREES

One kind of graph the topological-sort algorithm cannot handle is a graph with *cycles*. What's a cycle? It's a path that ends up where it started. In Figure 4.14 the path B-C-D-B forms a cycle. (Notice that A-B-C-A is not a cycle because you can't go from C to A.)

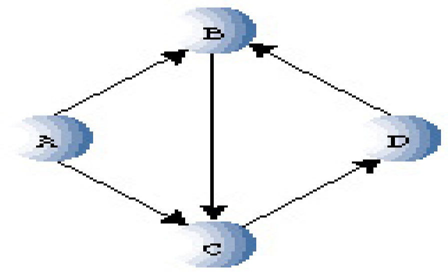


FIGURE 4.14 Graph with a cycle

A cycle models the Catch-22 situation (which some students claim to have actually encountered at certain institutions), where course B is a prerequisite for course C, C is a prerequisite for D, and D is a prerequisite for B.
A graph with no cycles is called a *tree*. The binary and multiway trees we saw earlier in this book are trees in this sense. However, the trees that arise in graphs are more general than binary and multiway trees, which have a fixed maximum number of child nodes. In a graph, a vertex in a tree can be connected to any number of other vertices, provided that no cycles are created.
A topological sort is carried out on a directed graph with no cycles. Such a graph is called a *directed, acyclic graph*, often abbreviated DAG.

C# CODE

Here's the C# code for the topo() method, which carries out the topological sort:

```
 public void topo()              // toplogical sort
   {
   int orig_nVerts = nVerts;     // remember how many verts

   while(nVerts > 0)             // while vertices remain,
     {
     // get a vertex with no successors, or -1
     int currentVertex = noSuccessors();
     if(currentVertex == -1)     // must be a cycle
       {
       Console.WriteLine("ERROR: Graph has cycles");
       return;
       }
     // insert vertex label in sorted array (start at end)
     sortedArray[nVerts-1] = vertexList[currentVertex].label;
```

```
        deleteVertex(currentVertex);// delete vertex
        }  // end while

    // vertices all gone; display sortedArray
    Console.Write("Topologically sorted order: ");
    for(int j=0; j<orig_nVerts; j++)
        Console.Write( sortedArray[j] );
    Console.WriteLine("");
    }  // end topo
```

The work is done in the while loop, which continues until the number of vertices is reduced to 0. Here are the steps involved:

1. Call noSuccessors() to find any vertex with no successors.
2. If such a vertex is found, put the vertex label at the end of sortedArray[] and delete the vertex from the graph.

If an appropriate vertex isn't found, the graph must have a cycle.
The last vertex to be removed appears first on the list, so the vertex label is placed in sortedArray starting at the end and working toward the beginning, as nVerts (the number of vertices in the graph) gets smaller.
If vertices remain in the graph but all of them have successors, the graph must have a cycle, and the algorithm displays a message and quits. Normally, however, the while loop exits, and the list from sortedArray is displayed, with the vertices in topologically sorted order.
The noSuccessors() method uses the adjacency matrix to find a vertex with no successors. In the outer for loop, it goes down the rows, looking at each vertex. For each vertex, it scans across the columns in the inner for loop, looking for a 1. If it finds one, it knows that that vertex has a successor, because there's an edge from that vertex to another one. When it finds a 1, it bails out of the inner loop so that the next vertex can be investigated.

Only if an entire row is found with no 1s do we know we have a vertex with no successors; in this case, its row number is returned. If no such vertex is found, –1 is returned. Here's the noSuccessors() method:

```
  public int noSuccessors()  // returns vert with no successors
    {                     // (or -1 if no such verts)
    bool  isEdge;  // edge from row to column in adjMat

    for(int row=0; row<nVerts; row++)  // for each vertex,
      {
      isEdge = false;                // check edges
      for(int col=0; col<nVerts; col++)
        {
        if( adjMat[row , col] > 0 )   // if edge to
          {                      // another,
          isEdge = true;
          break;                  // this vertex
          }                    //    has a successor
        }                      //    try another
      if( !isEdge )                // if no edges,
        return row;                //    has no successors
      }
    return -1;                     // no such vertex
    }  // end noSuccessors()
```

Deleting a vertex is straightforward except for a few details. The vertex is removed from the vertexList[] array, and the vertices above it are moved down to fill up the vacant position. Likewise, the row and column for the vertex are removed from the adjacency matrix, and the rows and columns above and to the right are moved down and to the left to fill the vacancies. This is carried out by the deleteVertex(), moveRowUp(), and moveColLeft() methods, which you can examine in the complete listing for topo.cs. It's actually more efficient to use the adjacency-list representation of the graph for this algorithm, but that would take us too far afield.

The Main() routine in this program calls on methods, similar to those we saw earlier, to create the same graph shown in Figure 4.10. The addEdge() method, as we noted, inserts a single number into the adjacency matrix because this is a directed graph. Here's the code for Main():

```
  public void Main(String[] args)
    {
    Graph theGraph = new Graph();
    theGraph.addVertex('A');    // 0
    theGraph.addVertex('B');    // 1
    theGraph.addVertex('C');    // 2
    theGraph.addVertex('D');    // 3
    theGraph.addVertex('E');    // 4
    theGraph.addVertex('F');    // 5
    theGraph.addVertex('G');    // 6
    theGraph.addVertex('H');    // 7

    theGraph.addEdge(0, 3);     // AD
    theGraph.addEdge(0, 4);     // AE
    theGraph.addEdge(1, 4);     // BE
    theGraph.addEdge(2, 5);     // CF
    theGraph.addEdge(3, 6);     // DG
    theGraph.addEdge(4, 6);     // EG
    theGraph.addEdge(5, 7);     // FH
    theGraph.addEdge(6, 7);     // GH

    theGraph.topo();            // do the sort
    }  // end main()
```

Once the graph is created, main() calls topo() to sort the graph and display the result. Here's the output:

```
  Topologically sorted order: BAEDGCFH
```

Of course, you can rewrite Main() to generate other graphs.

The Complete topological.cs Program

You’ve seen most of the routines in topological.cs already. Listing 4.4 shows the complete program.

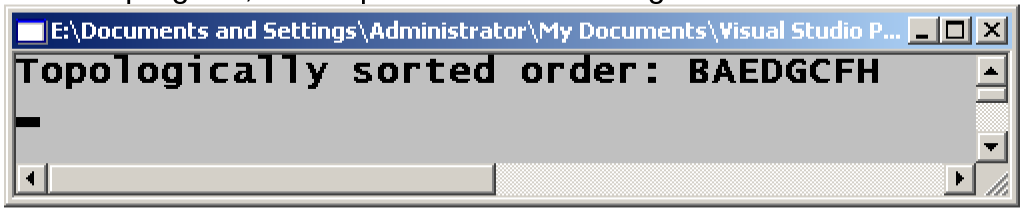
Listing 4.4 The topological.cs Program

```
using System;
        // topological.cs demonstrates topological sorting
namespace TopologicalSort
{
        class Vertex
        {
                public char label;        // label (e.g. 'A')

                public Vertex(char lab)   // constructor
                { label = lab; }
        }  // end class Vertex
        ////////////////////////////////////////////////////////////////
        class Graph
        {
                private  int MAX_VERTS = 20;
                private Vertex[] vertexList; // list of vertices
                private int[,] adjMat;      // adjacency matrix
                private int nVerts;         // current number of vertices
                private char [] sortedArray;
                // ------------------------------------------------------------
                public Graph()              // constructor
                {
                        vertexList = new Vertex[MAX_VERTS];
                        // adjacency matrix
                        adjMat = new int[MAX_VERTS , MAX_VERTS];
                        nVerts = 0;
                        for(int j=0; j<MAX_VERTS; j++)      // set adjacency
                                for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
                                        adjMat[j,k] = 0;
                        sortedArray = new char[MAX_VERTS];  // sorted vert labels
                }  // end constructor
                // ------------------------------------------------------------
                public void addVertex(char lab)
                {
                        vertexList[nVerts++] = new Vertex(lab);
                }
                // ------------------------------------------------------------
                public void addEdge(int start, int end)
                {
                        adjMat[start,end] = 1;
                }
```

```
// ------------------------------------------------------------
public void displayVertex(int v)
{
        Console.Write(vertexList[v].label);
}
// ------------------------------------------------------------
public void topo()  // toplogical sort
{
        int orig_nVerts = nVerts;  // remember how many verts

        while(nVerts > 0)  // while vertices remain,
        {
                // get a vertex with no successors, or -1
                int currentVertex = noSuccessors();
                if(currentVertex == -1)       // must be a cycle
                {
                        Console.WriteLine("ERROR: Graph has cycles");
                        return;
                }
                // insert vertex label in sorted array (start at end)
                sortedArray[nVerts-1] = vertexList[currentVertex].label;

                deleteVertex(currentVertex);  // delete vertex
        }  // end while

        // vertices all gone; display sortedArray
        Console.Write("Topologically sorted order: ");
        for(int j=0; j<orig_nVerts; j++)
                Console.Write( sortedArray[j] );
        Console.WriteLine("");
}  // end topo
// ------------------
public int noSuccessors()  // returns vert with no successors
{                     // (or -1 if no such verts)
        bool isEdge;  // edge from row to column in adjMat

        for(int row=0; row<nVerts; row++)  // for each vertex,
        {
                isEdge = false;                // check edges
                for(int col=0; col<nVerts; col++)
                {
                        if( adjMat[row,col] > 0 )   // if edge to
                        {                        // another,
                                isEdge = true;
                                break;                  // this vertex
                        }                        //    has a successor
                }                        //    try another
                if( !isEdge )                  // if no edges,
                    return row;                //    has no successors
            }
            return -1;                    // no such vertex
        }  // end noSuccessors()
        // ------------------
        public void deleteVertex(int delVert)
        {
            if(delVert != nVerts-1)      // if not last vertex,
            {                        // delete from vertexList
                for(int j=delVert; j<nVerts-1; j++)
                    vertexList[j] = vertexList[j+1];
                // delete row from adjMat
                for(int row=delVert; row<nVerts-1; row++)
                    moveRowUp(row, nVerts);
                // delete col from adjMat
                for(int col=delVert; col<nVerts-1; col++)
                    moveColLeft(col, nVerts-1);
            }
            nVerts--;                  // one less vertex
        }  // end deleteVertex
        // ------------------
        private void moveRowUp(int row, int length)
        {
            for(int col=0; col<length; col++)
                adjMat[row,col] = adjMat[row+1,col];
        }
        // ------------------
        private void moveColLeft(int col, int length)
        {
            for(int row=0; row<length; row++)
                adjMat[row,col] = adjMat[row,col+1];
        }
        // ------------------------------------------------------------
    }  // end class Graph
    ////////////////////////////////////////////////////////////////
    class TopoApp
    {
    static void Main(string[] args)
    {
        Graph theGraph = new Graph();
        theGraph.addVertex('A');    // 0
        theGraph.addVertex('B');    // 1
        theGraph.addVertex('C');    // 2
        theGraph.addVertex('D');    // 3
```

```
                theGraph.addVertex('E');    // 4
                theGraph.addVertex('F');    // 5
                theGraph.addVertex('G');    // 6
                theGraph.addVertex('H');    // 7

                theGraph.addEdge(0, 3);     // AD
                theGraph.addEdge(0, 4);     // AE
                theGraph.addEdge(1, 4);     // BE
                theGraph.addEdge(2, 5);     // CF
                theGraph.addEdge(3, 6);     // DG
                theGraph.addEdge(4, 6);     // EG
                theGraph.addEdge(5, 7);     // FH
                theGraph.addEdge(6, 7);     // GH

                theGraph.topo();            // do the sort
                Console.ReadLine();
        }
      }  // end class TopoApp
        ////////////////////////////////////////////////////////////////
}
```

Run the program, the output is as the following.



```
Topologically sorted order: BAEDGCFH
```

In the next chapter, we'll see what happens when edges are given a weight as well as a direction.

Summary

- Graphs consist of vertices connected by edges.
- Graphs can represent many real-world entities, including airline routes, electrical circuits, and job scheduling.
- Search algorithms allow you to visit each vertex in a graph in a systematic way. Searches are the basis of several other activities.
- The two main search algorithms are depth-first search (DFS) and breadth-first search (BFS).
- The depth-first search algorithm can be based on a stack; the breadth-first search algorithm can be based on a queue.
- A minimum spanning tree (MST) consists of the minimum number of edges necessary to connect all a graph's vertices.
- A slight modification of the depth-first search algorithm on an un-weighted graph yields its minimum spanning tree.
- In a directed graph, edges have a direction (often indicated by an arrow).

- A topological sorting algorithm creates a list of vertices arranged so that a vertex A precedes a vertex B in the list if there’s a path from A to B.
- A topological sort can be carried out only on a DAG, a directed, acyclic (no cycles) graph.
- Topological sorting is typically used for scheduling complex projects that consist of tasks contingent on other tasks.