Chapter 13
Graphs

- Introduction to Graphs
- Searches
- Minimum Spanning Trees
- Topological Sorting with Directed Graphs
- Connectivity in Directed Graphs
- Summary

Graphs - Overview

- One of the most versatile structures used in computer programming.
- The sorts of problems that graphs can help solve are generally quite different from those we’ve dealt with thus far.
- If dealing with general kinds of data storage problems, probably won’t need a graph.
  - but for some problems—and they tend to be interesting ones—a graph is indispensable.
- This chapter covers the algorithms associated with unweighted graphs
- Next chapter: More complicated algorithms associated with weighted graphs.

Introduction to Graphs

- Graphs are data structures rather like trees.
- In a mathematical sense, a tree is a kind of graph.
- In computer programming, graphs are used in different ways than trees.
- The data structures examined previously have an architecture dictated by the algorithms used on them.
  - For example, a binary tree is shaped the way it is because that shape makes it easy to search for data and insert new data.
  - The edges in a tree represent quick ways to get from node to node.
- Graphs often have a shape dictated by a physical problem.
  - For example, a graph may represent cities and edges may represent airline flight routes between the cities.
  - Another more abstract example is a graph representing the individual tasks necessary to complete a project.
    - In the graph, nodes may represent tasks, while directed edges (with an arrow at one end) indicate which task must be completed before another.
    - In both cases, the shape of the graph arises from the specific real-world situation.

Definitions

- Figure 13.1-a shows a simplified map of the freeways in the vicinity of San Jose, California.
- Figure 13.1-b shows a graph that models these freeways.
- In the graph,
  - circles represent freeway interchanges
  - straight lines connecting the circles represent freeway segments.
  - The circles are vertices, and the lines are edges.
  - The vertices are usually labeled in some way—often, as shown here, with letters of the alphabet.
  - Each edge is bounded by the two vertices at its ends.

Introduction to Graphs

- When discussing graphs, nodes are called vertices (the singular is vertex).
  - Probably because the nomenclature for graphs is older than that for trees, having arisen in mathematics centuries ago.
  - Trees are more closely associated with computer science.

Definitions

- The graph doesn’t attempt to reflect the geographical positions shown on the map:
  - it shows only the relationships of the vertices and the edges—that is, which edges are connected to which vertex.
  - It doesn’t concern itself with physical distances or directions.
  - Also, one edge may represent several different route numbers,
    - as in the case of the edge from I to H, which involves routes 101, 84, and 280.
  - It’s the connectedness (or lack of it) of one intersection to another that’s important, not the actual routes.
Introduction to Graphs

Definitions

Adjacency

- Two vertices are said to be adjacent to one another if they are connected by a single edge.
- In Figure 13.1, vertices I and D are adjacent, but vertices I and F are not.
- The vertices adjacent to a given vertex are sometimes said to be its neighbors.
  - For example, the neighbors of G are I, H, and F.

Figure 13.1: Road map and graph

Paths

- A path is a sequence of edges.
- Figure 13.1 shows a path from vertex B to vertex J that passes through vertices A and E.
  - We can call this path BAEJ.
- There can be more than one path between two vertices;
  - another path from B to J is BCDJ.

Figure 13.1: Road map and graph

Connected Graphs

- A graph is said to be connected
  - if there is at least one path from every vertex to every other vertex,
  - as in the graph in Figure 13.2-a.
- However, if “You can’t get there from here” (as Vermont farmers traditionally tell city slickers who stop to ask for directions), the graph is not connected, as in Figure 13.2-b.
- There can be more than one path between two vertices;
  - another path from B to J is BCDJ.

Figure 13.2: Connected and non-connected graphs

Directed and Weighted Graphs

- The graphs in Figures 13.1 and 13.2 are non-directed graphs.
  - That means that the edges don’t have a direction; you can go either way on them.
  - Thus you can go from vertex A to vertex B, or from vertex B to vertex A, with equal ease.
  - (This models freeways appropriately, because you can usually go either way on a freeway.)
- However, graphs are often used to model situations in which you can go in only one direction along an edge;
  - from A to B but not from B to A, as on a one-way street.
  - Such a graph is said to be directed.
- The allowed direction is typically shown with an arrowhead at the end of the edge.
- In some graphs, edges are given a weight,
  - a number that can represent the physical distance between two vertices,
  - or the time it takes to get from one vertex to another,
  - or how much it costs to travel from vertex to vertex (on airline routes, for example).
  - Such graphs are called weighted graphs. (Next chapter)

Figure 13.1: Road map and graph

Introduction to Graphs

Historical Note

- One of the first mathematicians to work with graphs was Leonhard Euler in the early 18th century.
  - The bridges in the town of Königsberg, Poland.
  - The problem was to find a way to walk across all seven bridges without re-crossing any of them.
  - It turned out that there is no such path.
- The key to his solution was to represent the problem as a graph,
  - with land areas as vertices and bridges as edges,
  - as shown in Figure 13.3-a.
- Perhaps the first example of a graph being used to represent an problem in the real world.
  - Represent graphs by using a computer. What sort of software structures are appropriate to model a graph?
  - We’ll look at vertices first, and then at edges.

Figure 13.3: The bridges of Königsberg

Introduction to Graphs

Representing a Graph in a Program

Vertices

- Simply number the vertices 0 to N–1 (where N is the number of vertices).
  - Wouldn’t need any sort of variable to hold the vertices,
    - because their usefulness would result from their relationships with other vertices.
  - In most situations, a vertex represents some real-world object, and the object must be described using data items.
    - If a vertex represents a city in an airline route simulation, for example,
      - it may need to store the name of the city, its altitude, its location,
        - and other such information.
  - Convenient to represent a vertex by an object of a vertex class.
    - Our example programs store only a letter (like A), used as a label for identifying the vertex, and a flag for use in search algorithms, as well see later.

Figure 13.3: The bridges of Königsberg
**Introduction to Graphs**

**Representing a Graph in a Program**

Here's how the `Vertex` class looks:

```java
public class Vertex {
    public char label; // e.g. "A"
    public boolean wasVisited;
    public Vertex(char lab) { // constructor
        label = lab;
        wasVisited = false;
    }
    // end class Vertex
}
```

• Vertex objects can be placed in an array and referred to using their index number.
• In our examples we'll store them in an array called `vertexList`.
• The vertices might also be placed in a list or some other data structure.
• Whatever structure is used, this storage is for convenience only.
• For this, we need another mechanism.

**The Adjacency Matrix**

- An adjacency matrix is a two-dimensional array
  - in which the elements indicate whether an edge is present between two vertices.
- If a graph has N vertices, the adjacency matrix is an N x N array.
- Table 13.1 shows the adjacency matrix for the graph in Figure 13.2-a.

Table 13.1: Adjacency Matrix

```
A B C D
A 0 1 1 1
B 1 0 0 1
C 1 0 0 0
D 1 1 1 0
```

Figure 13.2-a: Connected and non-connected graphs

**Introduction to Graphs**

**Representing a Graph in a Program**

**The Adjacency List**

- The vertices are used as headings for both rows and columns.
- An edge between two vertices is indicated by a 1; the absence of an edge is a 0.
- (You could also use Boolean true/false values.)
- As you can see, vertex A is adjacent to all three other vertices, B is adjacent to A and D, C is adjacent only to A, and D is adjacent to A and B.
- In this example, the "connection" of a vertex to itself is indicated by 0, so the diagonal from upper-left to lower-right, A-A to D-D, which is called the identity diagonal, is all 0s.

Table 13.2: Adjacency List

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 13.2-b: Connected and non-connected graphs

**Introduction to Graphs**

**Representing a Graph in a Program**

**The Adjacency List**

- The other way to represent edges is with an adjacency list.
- The list in an adjacency list refers to a linked list of the kind we examined in Chapter 6, "Recursion."
- Actually, an adjacency list is an array of lists (or a list of lists).
- Each individual list shows what vertices a given vertex is adjacent to.
- Table 13.2 shows the adjacency lists for the graph of Figure 13.2-a.

Table 13.2: Adjacency List

<table>
<thead>
<tr>
<th>V</th>
<th>Neighbor List</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B, B, D</td>
</tr>
<tr>
<td>B</td>
<td>A, C</td>
</tr>
<tr>
<td>C</td>
<td>A, D</td>
</tr>
<tr>
<td>D</td>
<td>A, C</td>
</tr>
</tbody>
</table>

Figure 13.2-c: Connected and non-connected graphs
Introduction to Graphs

Representing a Graph in a Program

The Adjacency List

- In this table, the -> symbol indicates a link in a linked list.
- Each link in the list is a vertex.
- Here the vertices are arranged in alphabetical order in each list.
- Although that's not really necessary.

- Don't confuse the contents of adjacency lists with paths.
- The adjacency list shows which vertices are adjacent to that is, one edge away from a given vertex, not paths from vertex to vertex.
- Later we'll discuss when to use an adjacency matrix as opposed to an adjacency list.

- The workshop applets shown in this chapter all use the adjacency matrix approach, but sometimes the list approach is more efficient.

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Containing Adjacent Vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B, C</td>
</tr>
<tr>
<td>B</td>
<td>A, C</td>
</tr>
<tr>
<td>C</td>
<td>A, D</td>
</tr>
<tr>
<td>D</td>
<td>A, B, C</td>
</tr>
</tbody>
</table>

Table 13.2: Adjacency List

Figure 13.4-a: A -> BD

Figure 13.4-b: A -> AC

Introduction to Graphs

Adding Vertices and Edges to a Graph

- To add a vertex to a graph,
  - you make a new Vertex object with new and insert it into your vertex array, vertexList.
- In a real-world program a vertex might contain many data items,
  - but for simplicity we'll assume that it contains only a single character.
  - Thus the creation of a vertex looks something like this:

```java
Vertex newVertex = new Vertex('F');
```

- This inserts a vertex F, where 'F' is the number of vertices currently in the graph.

- To add an edge to a graph depends on whether you're using an adjacency matrix or adjacency lists to represent the graph.
- Using an adjacency matrix: To add an edge between vertices 1 and 3.
  - These numbers correspond to the array indices in vertexList, where the vertices are stored.
  - When you first created the adjacency matrix adjMat, you filled it with 0s. To insert the edge, you say

```java
adjMat[1][3] = 1;
```

- Using an adjacency list: Add 1 to the list for 3, and a 3 to the list for 1.

Introduction to Graphs

Searches

- Finding which vertices can be reached from a specified vertex.
  - For example, imagine trying to find out how many towns in the United States can be reached by passenger train from Kansas City (assuming that you don't mind changing trains).
  - Some towns could be reached.
  - Others couldn't be reached because they didn't have passenger rail service.
  - Possibly others couldn't be reached, even though they had rail service, because their rail system (the narrow-gauge Hayfork-Hicksville RR, for example) didn't connect with the standard-gauge lines you started on or any of the lines that could be reached from your line.

- Here's another situation in which you might need to find all the vertices reachable from a specified vertex.
  - Imagine that you're designing a printed circuit board, like the ones inside your computer.
    - Various components—mostly ICs—are placed on the board, with pins from the ICs protruding through holes in the board.
    - The ICs are soldered in place, and their pins are electrically connected to other pins by traces—thin metal lines applied to the surface of the circuit board.
    - As shown in Figure 13.4.

- On a circuit board there are many electrical circuits that aren't connected to each other, so the graph is by no means a connected electrical circuit.

- Assume that you've created such a graph.
  - Now you need an algorithm that provides a systematic way to start at a specified vertex, and then move along edges to other vertices, in such a way that when it's done you are guaranteed that it has visited every vertex that's connected to the starting vertex.

- Here, as it did in Chapter 8, “Binary Trees,” when we discussed binary trees, visit means to perform some operation on the vertex, such as displaying it.

- There are two common approaches to searching a graph:
  - depth-first search (DFS)
  - breadth-first search (BFS)

- Both will eventually reach all connected vertices.
  - The difference is that the DFS is implemented with a stack whereas the BFS is implemented with a queue.
  - These mechanisms result, as we'll see, in the graph being searched in different ways.

- In a graph, each pin might be represented by a vertex, and each trace by an edge.

- On a circuit board there are many electrical circuits that aren't connected to each other, so the graph is by no means a connected electrical circuit.

- During the design process, therefore, it may be genuinely useful to create a graph and use it to find which pins are connected to the same electrical circuit.

- Assume that you've created such a graph.
  - Now you need an algorithm that provides a systematic way to start at a specified vertex, and then move along edges to other vertices, in such a way that when it's done you are guaranteed that it has visited every vertex that's connected to the starting vertex.

- Here, as it did in Chapter 8, “Binary Trees,” when we discussed binary trees, visit means to perform some operation on the vertex, such as displaying it.

- There are two common approaches to searching a graph:
  - depth-first search (DFS)
  - breadth-first search (BFS)

- Both will eventually reach all connected vertices.
  - The difference is that the DFS is implemented with a stack whereas the BFS is implemented with a queue.
  - These mechanisms result, as we'll see, in the graph being searched in different ways.

- In a graph, each pin might be represented by a vertex, and each trace by an edge.

- On a circuit board there are many electrical circuits that aren't connected to each other, so the graph is by no means a connected electrical circuit.

- During the design process, therefore, it may be genuinely useful to create a graph and use it to find which pins are connected to the same electrical circuit.

- Assume that you've created such a graph.
  - Now you need an algorithm that provides a systematic way to start at a specified vertex, and then move along edges to other vertices, in such a way that when it's done you are guaranteed that it has visited every vertex that's connected to the starting vertex.

- Here, as it did in Chapter 8, “Binary Trees,” when we discussed binary trees, visit means to perform some operation on the vertex, such as displaying it.

- There are two common approaches to searching a graph:
  - depth-first search (DFS)
  - breadth-first search (BFS)

- Both will eventually reach all connected vertices.
  - The difference is that the DFS is implemented with a stack whereas the BFS is implemented with a queue.
  - These mechanisms result, as we'll see, in the graph being searched in different ways.
Searches - Depth-First Search

• The DFS uses a stack to remember where it should go when it reaches a dead end.

An Example

• We'll discuss the idea behind the DFS in relation to Figure 13.5.
• The numbers in this figure show the order in which the vertices are visited.

To carry out the DFS,
• 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.
  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.
  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.
• If H 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 3.
  Rule 3:
  If you can't follow Rule 1 or Rule 2, you're finished.
• Table 13.3 shows how the stack looks in the various stages of this process, as applied to Figure 13.5.
• 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.
• You might say that the DFS algorithm likes to get as far away from the starting point as possible, and then 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 depth-first search comes from.

An Analogy

• 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.
– All 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 Graph Workshop Applet and DFS

- With the DFS button in the Graph 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 13.5: The Graph Workshop applet

Java Code

- A key to the DFS algorithm is being able to find the vertices that are unvisited and adjacent to a specified vertex.

- The adjacency matrix is the key.
- How do you do this?
- You can then check whether this vertex is unvisited.
- 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 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:

```java
public int getAdjUnvisitedVertex(int v) // returns an unvisited vertex adjacent to v
        {
            for(int j=0; j<nVerts; j++) // reset flags
                vertexList[j].wasVisited = false;

            int v = getAdjUnvisitedVertex( theStack.peek() );
            theStack.push(v); // push it
            displayVertex(v); // display it
            vertexList[v].wasVisited = true; // mark it
        }
```

- 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.
  3. 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.

The dfs() method of the Graph class, which actually carries out the depth-first search.

- As shown in Figure 13.7.
- Clicking View again switches you back to the graph.
- 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.
- If you use the algorithm on an unconnected graph, it will find only those vertices that are connected to the starting vertex.

Figure 13.6: Adjacency matrix view in GraphNSearches.pdf

- (It warns you before it does this.)

```java
public void dfs() // depth-first search
        {
            public void dfs() // depth-first search
            {
                vertexList[0].wasVisited = true; // mark it
                displayVertex(0); // display it
                while( !theStack.isEmpty() ) // until stack empty,
                    {
                        int v = theStack.pop(); // pop a new one
                        if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
                            {
                                vertexList[j].wasVisited = true; // mark it
                                displayVertex(j); // display it
                                theStack.push(0); // push it
                            }
                    }
            }
```

- Figure 13.6 shows the graph of Figure 13.5 as it looks when created using the applet.

- The adjacency matrix for the graph you've made, as shown in Figure 13.7.
- Clicking View again switches you back to the graph.
- 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.
- If you use the algorithm on an unconnected graph, it will find only those vertices that are connected to the starting vertex.
Searches - Breadth-First Search

The dfs.java Program

- It includes the dfs() method.
- It includes a version of the StackX class from Chapter 4, "Stacks and Queues."

See Listing 13.1

Searches - Breadth-First Search

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

**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.

- In DFS, 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 (BFS), 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 away.
- This kind of search is implemented using a queue instead of a stack.

**An Example**

- Figure 13.9 BFS is used.
- Again, the numbers indicate the order in which the vertices are visited.

Given the same graph as in dfs.java (Figure 13.8), the output from dfs.java is now

```java
Visits: ABDCE
```

Figure 13.9 BFS is used.

```java
Queue (Front to Rear) Event
```

- 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.

```java
Given the same graph as in dfs.java (Figure 13.8), the output from dfs.java is now

```java
Visits: ABDCE
```

**Java Code**

The dfs.java method 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.

```
Given the same graph as in dfs.java (Figure 13.8), the output from dfs.java is now

```java
Visits: ABDCE
```

```
```

```java
Given the same graph as in dfs.java (Figure 13.8), the output from dfs.java is now

```java
Visits: ABDCE
```

**Searches - Breadth-First Search (BFS)**

- First visit all the vertices one edge (plane flight) away from the starting point as you visit them.
- Now you've visited A, B, C, D, and E. At this point the queue (from front to rear) contains BCDE.
- 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 13.4 shows this sequence.
- 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 DFS, where the contents of the stack always indicate you last visited the starting point to the current vertex.
- The nodes are visited in the order ABCDEFGHI.

**Searches - Breadth-First Search**

Table 13.4: Queue Contents During Breadth-First Search

<table>
<thead>
<tr>
<th>Vertex</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Searches - Breadth-First Search**

The Graph Workshop Applet and BFS

- Try out a BFS using the BFS button.
- Again, you can experiment with the graph of Figure 13.9, or you can make up your own.
- Notice the similarities and the differences of the BFS compared with the DFS.
- You can think of the BFS as:
  - proceeding like ripples widening when you drop a stone in water
  - the influenza virus carried by air travelers from city to city.
  - The nodes are visited in the order ABCDEFGHI.
  - At each moment, the queue contains the vertices that have been visited but whose neighbors have not yet been fully explored.

**An Example**

- Figure 13.9 BFS is used.
- Again, the numbers indicate the order in which the vertices are visited.

```java
Queue (Front to Rear) Event
```

- The contents of the stack always indicate you last visited the starting point to the current vertex.

```java
Given the same graph as in dfs.java (Figure 13.8), the output from dfs.java is now

```java
Visits: ABDCE
```

**Searches - Breadth-First Search**

- Given the same graph as in dfs.java (Figure 13.8), the output from dfs.java is now

```java
Visits: ABDCE
```

**Graphs Workshop Applet and BFS**

- You can think of the BFS as:
  - 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.
  - 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.

```java
Given the same graph as in dfs.java (Figure 13.8), the output from dfs.java is now

```java
Visits: ABDCE
```

**Java Code**

The dfs.java method 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.

```
Given the same graph as in dfs.java (Figure 13.8), the output from dfs.java is now

```java
Visits: ABDCE
```

**Searches - Breadth-First Search**

The dfs.java method 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.

```
Given the same graph as in dfs.java (Figure 13.8), the output from dfs.java is now

```java
Visits: ABDCE
```
Minimum Spanning Trees

- Designing a printed circuit board (Figure 13.4), want to be sure having used the minimum number of traces.
  - That is, 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.

- 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 13.10-a shows five vertices with an excessive number of edges, while Figure 13.10-b shows the same vertices with the minimum number of edges necessary to connect them.
  - This constitutes a minimum spanning tree.

The graph that results is the one shown in Figure 13.10-a. When the mst() method has done its work, only four edges are left, as shown in Figure 13.10-b. Here's the output from the mst() program:
Minimum spanning tree: AB BC CD DE

• 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.

• Repeatedly clicking the Tree button in the GraphN Workshop algorithm will create a minimum spanning tree for any graph you create.

• 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.

• 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 if it's these edges that make up the minimum spanning tree.

- In the main() part of the mst.java 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
```

• For example, Figure 13.10-a shows five vertices with an excessive number of edges, while Figure 13.10-b shows the same vertices with the minimum number of edges necessary to connect them.

• Such extra connections would take up extra room and make other circuits more difficult to lay out.

• For example, Figure 13.10-a shows five vertices with an excessive number of edges, while Figure 13.10-b shows the same vertices with the minimum number of edges necessary to connect them.

• This constitutes a minimum spanning tree.

• Repeatedly clicking the Tree button in the GraphN Workshop algorithm will create a minimum spanning tree for any graph you create.

• 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.

• 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 if it's these edges that make up the minimum spanning tree.

- In the main() part of the mst.java 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
```

• The result would be a graph with the minimum number of edges necessary to connect the vertices.

• For example, Figure 13.10-a shows five vertices with an excessive number of edges, while Figure 13.10-b shows the same vertices with the minimum number of edges necessary to connect them.

• This constitutes a minimum spanning tree.

Minimum Spanning Trees - Java Code for the Minimum Spanning Tree

```
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
        System.out.print(" ");
    }
// 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()
```

Minimum Spanning Trees - Java Code for the Minimum Spanning Tree

```
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
        System.out.print(" ");
    }
// 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()
```

Minimum Spanning Trees - Java Code for the Minimum Spanning Tree

```
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
        System.out.print(" ");
    }
// 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()
```
Minimum Spanning Trees

Java Code for the Minimum Spanning Tree

- 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.java` Program

- Similar to `dfs.java`, except for the `mst()` method and the graph created in `main()`.
  - See Listing 13.3.

Topological Sorting with Directed Graphs

An Example: Course Prerequisites

- Some courses have prerequisites — other courses that must be taken first.
- Taking certain courses may be a prerequisite to obtaining a degree in a certain field.
- Figure 13.11 shows a somewhat fanciful arrangement of courses necessary for graduating with a degree in mathematics.
  - 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

- 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, 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, `adjMat[start][end] = 1;`

Topological Sorting

- Imagine that you make a list of all the courses necessary for your degree, using Figure 13.11 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: `CFCBEADGH`.
    - This also satisfies all the prerequisites.
- There are many other possible orderings as well.
- When we use an algorithm to generate a topological sort, the approach we take and the details of the code determine which of various valid sortings are generated.
Topological Sorting with Directed Graphs

Topological Sorting

• Topological sorting can model other situations:
  – Job scheduling
    • In building a car, you want to arrange things:
      – brakes → wheels
      – 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.
  – 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.

  Figure 13.14: Graph with a cycle

  // A topological sort is carried out on a directed graph with no cycles.
  // A graph with no cycles is called a directed, acyclic graph, often abbreviated DAG.

  ```java
  public void topo() // topological sort
  {
    // vertices all gone; display sortedArray
    System.out.println("");
    for(int j=0; j<orig_nVerts; j++)
      System.out.print("Topologically sorted order: ");
    System.out.println("");
    // insert vertex label in sorted array (start at end)
    for(int j=nVerts-1; j>=0; j--)
    {
      // vertices all gone; display sortedArray
      System.out.print( sortedArray[j] );
      if(currentVertex == -1) // must be a cycle
        System.out.println("ERROR: Graph has cycles");
    }
  }

  int orig_nVerts = nVerts; // remember how many verts
  int currentVertex = noSuccessors(); // get a vertex with no successors, or -1
  // insert vertex label in sorted array (start at end)

  step 1: 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 13.11, the only vertex with no successors is H.

  step 2: Delete this vertex from the graph, and insert its label at the beginning of the 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.

  Figure 13.11: Course prerequisites

  It’s a path that ends up where it started.
  – 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 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.

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

  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 13.14 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.).

  • 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, 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.

  The top() method carries out the topological sort:

  ```java
  public void top() // topological sort
  {
    int orig_vverts = vverts; // remember how many verts
    while(vverts > 0) // while vertices remain,
    { }

    System.out.println("ERROR: Graph has cycles");
  }

  // vertices all gone; display sortedArray
  System.out.println("Topologically sorted order: ");
  for(int i=0; i<orig_vverts; i++)
  System.out.println(sortedArray[i]);
  // end top

  // See next slides for explanations

  ```
Topological Sorting with Directed Graphs - Java Code

The work is done in the while loop, which continues until the number of vertices is reduced to 0. 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.
3. If no appropriate vertex isn't found, the graph must have a cycle.
4. 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.
5. If vertices remain in the graph but all of them have successors, the graph must have a cycle.
   – Normally, however, the while loop exits, and the list from sortedArray[] is displayed, with the vertices in topologically sorted order.

Topological Sorting with Directed Graphs - Java Code

public int noSuccessors() // returns vert with no successors
{ // (or -1 if no such vertex)
    boolean isEdge; // edge from row to column in adjMat
    for(int row=0; row<nVerts; row++) // for each vertex,
    { // (or -1 if no such 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
            } // try another
        } // try another
        if( !isEdge ) // if no edges,
            return row; // has no successors
    } // end for
    return -1; // no such vertex
} // end noSuccessors()
Connectivity in Directed Graphs

• Can’t just do an arbitrary BFS or DFS
  - Connectivity depends on starting node, i.e., “what can you reach from node X?”
  - Do DFS from every vertex!

• Alternative: develop connectivity matrix from adjacency matrix
  - Transitive closure of adjacency matrix
  - If L -> M and M -> N, L -> N

Warshall’s Algorithm

• For all rows y,
  - For all columns x in row y,
  • If any value (x, y) is 1,
  • For all rows z in column y,
    - If (y, z) is 1, then (x, z) should be 1
  • That’s it!
    - Remember array references are “backwards” [y][x]
• Yes, this actually works in one pass all the holes are filled
• What’s the complexity of this algorithm?

Summary (I)

• 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 DFS algorithm can be based on a stack; the BFS algorithm can be based on a queue.

Summary (II)

• 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 DFS algorithm on an unweighted 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.