Graph

Graph

- One of the most **versatile structures**.
- Dictated by a **physical** or abstract **problem. Example of such problem can be:**
  - **Nodes** in a graph may represent **cities**, while **edges** may represent **airline flight routes** between the cities.
  - Individual **tasks** necessary to complete a project. In the graph, nodes may represent **tasks**, while **directed edges** indicate which task must be completed before another.
  - **Internet routes**.

Graph Terminology

Graph

- **Adjacency**:
  - Two vertices are said to be adjacent to one another if they are connected by a single edge.
- **Paths**:
  - A path is a sequence of edges.
- **Connected graphs:**
  - A graph is said to be connected if there is at least one path from every vertex to every other vertex.
- **Directed and weighted graphs:**

Representing a graph in a program

Graph

- **Vertices**:
- It is usually convenient to represent a vertex by an object of a vertex class.

```
class vertex
{
   public char label;
   public Boolean wasvisited;
   public vertex(char lab)
  {
            label = lab;
            wasvisited = false;
   }
}
```

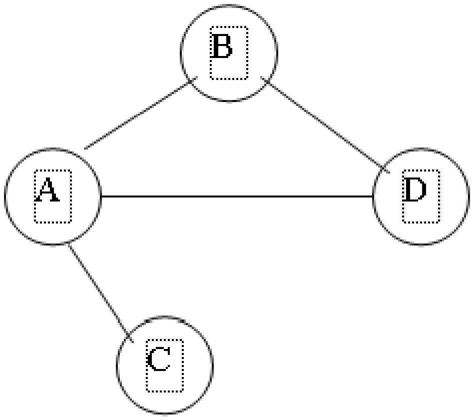
Graph

- vertex **object** can be **placed** in an **array** and referred to using their **index number**. The **vertices** might also be placed in a **list** or some **other data structure**.

Graph

- **Edges:**
  - In a **binary tree, each node** has a **maximum** of **two children,** but in a **graph** each **vertex** may be **connected** to an **arbitrary number** of other **vertices.**
  - To model this sort of **free-form organization**, two methods are commonly used for graphs:
    - **Adjacency Matrix**
    - **Adjacency List**
  - **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 **NxN** array. See the following example:

Graph



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

Graph

- **The adjacency list:**
- The list in **adjacency list** refers to a **linked list**.
- Each **individual list** shows what **vertices** a given vertex is **adjacent** to. Following is the **adjacency list** for the above **graph**:

Vertex	List Containing Adjacent Vertex
A	B→C→D
B	A→D
C	A
D	A→B

Adding Vertices and Edges to a Graph

Graph

- To **add** a **vertex** to a **graph**, you make anew **vertex object** and **Insert** it into your vertex **array**, **vertexList**.
- The creation of a vertex looks something like this:
  - **vertexList[nVerts**++] = **new vertex**('F');
- To **insert** the **edge**, you say:
  - **adjMat[1][3] = 1;**
  - **adjMat[3][1] = 1;**

The Graph Class

Graph

- The **following code** shows a class contains **methods** for **creating** a **vertex** list and an **adjacency matrix**, and for **adding vertices** and **edges** to a **graph** object:

```
class graph
{
    private final int max_verts = 20;
    private vertex vertexlist[];
    private int adjMat[][];
    private int nverts;
    public graph()
    {
        vertexlist = new vertex[max_verts];
        adjMat = new[max_verts][max_verts];
        nverts = 0;
        for(int j=0; j<max_verts; j++)
            for(int k=0; k<max_verts; k++)
            adjMat[j][k] =0;
    }
```

Graph

```
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)
{
    System.out.print(vertexlist[v].label);
}
}
```

Searches

Graph

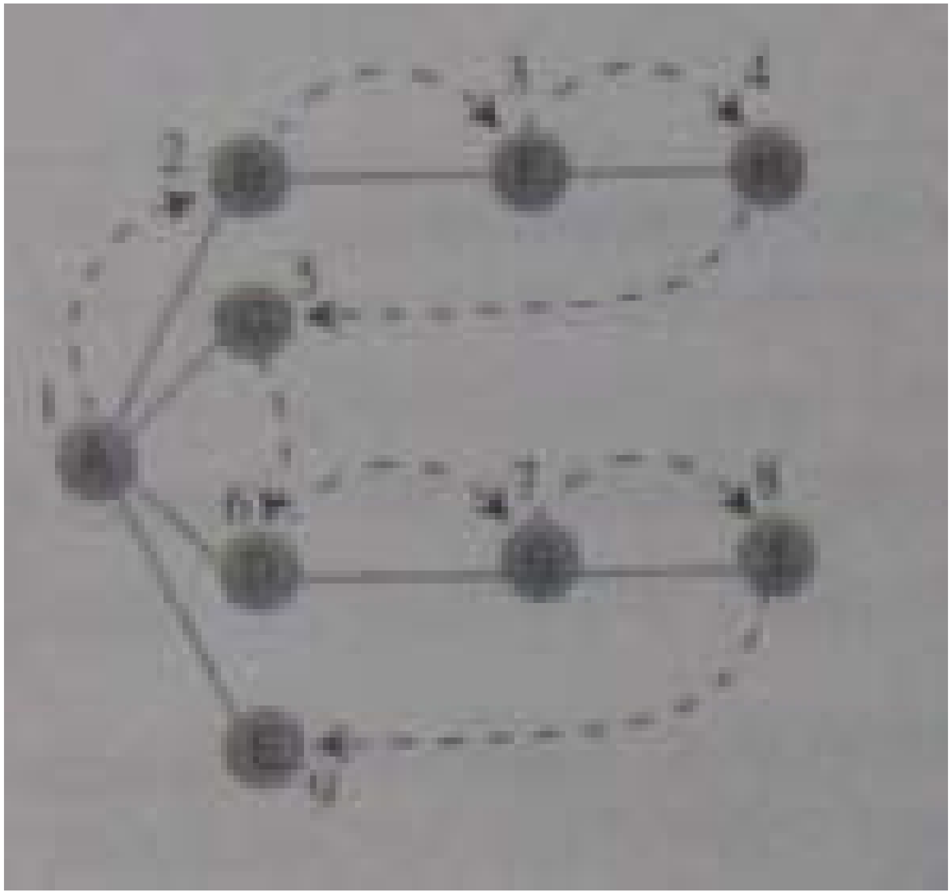
- One of the **most fundamental operations** to perform on a graph is **finding which vertices** can be **reached** from a **specified vertex**.
- There are **two common approaches** to searching a graph:
  - Depth-first search (**DFS**)
  - Breadth-first search (**BFS**)
- **Both** will eventually **reach** all **connected vertices**.
- The **DFS** is implemented with a **stack**, whereas the **BFS** is implemented with a **queue**.

Depth-first search

Graph

The **depth-first** search uses a **stack** to **remember where** it should go when it **reaches** a **dead end**.

Example: consider the following **figure**:



Graph

- To **carry** out the **DFS**:
  - **Pick** a **starting point**, in this case, vertex **A**.
  - You then **do 3 things**:
    - **visit** this vertex,
    - **push** it onto a stack, and
    - **mark** it.
  - **Next** you go to any **vertex adjacent** to **A** that has **not** yet been **visited**.
  - You **visit B**, **mark** it, and **push** it on the stack. Now what? **You are at B**, and you do the **same thing** as before: go to an **adjacent vertex** that has **not** been **visited**. This lead you to **F**. We can call this process **Rule 1**.

Graph

- **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 is where is **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**. F has **no unvisited** adjacent vertices, so you **pop** it, **and also B**. Now **only A** is **left** on the **stack**.
- **Rule 3:** If you can not **follow Rule 1 or Rule 2**, you are done.

Graph

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	
Done	

Graph

- 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**.

Java Code

Graph

- 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 have found what you want, the **next vertex** to **visit**. If **no** vertices on the row are **simultaneously 1** (**adjacent**) and also unvisited, there are no **unvisited** vertices adjacent to the specified vertex.
- We **put** the **code** for this process in the **getadjunvisitedvertex**() **method**:

Graph

```
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;
}
```

Graph

- 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:
    - It examines the vertex at the top of the stack, using peek().
    - It tries to find an unvisited neighbors of this vertex.
    - If it does not find one, it pops the stack.
- If it finds such a vertex, it visits that vertex and pushes it onto the stack.

Graph

```
public void dfs()
{
      vertexlist[0].wasvisited = true;
      displayvertex(0);
      thestack.push(0);
      while ( !thestack.isempty() ) {
                int v = getadjunvisitedvertex ( thestack.peek() );
                if ( v == -1)  // if no such vertex,
                                thestack.pop();  // pop a new one
                else
                                {
                                vertexlist[v].wasvisited = true;
                                displayvertex(v);
                                thestack.push(v);
                                }
      }

      for (int j=0; j<nverts; j++)
                vertexlist[j].wasvisited = false;
}
```

Graph

- At the **end** of **dfs**(), we **reset** all the **wasvisited flags** so we will be ready to run **dfs**() again later. The **stack** should already be **empty**, so it does not need to be **reset**.
- Now we have **all** the **pieces** of the **graph** class we need. Here is some code that **creates** a graph **object**, adds some vertices and edges to it, and then performs a **depth-first search**:

Graph

```
Graph thegraph = new graph();
Thegraph.addvertex(‘A’);
Thegraph.addvertex(‘B’);
Thegraph.addvertex(‘C’);
Thegraph.addvertex(‘D’);
Thegraph.addvertex(‘E’);
Thegraph.addedge(0, 1);
Thegraph.addedge(1, 2);
Thegraph.addedge(0, 3);
Thegraph.addedge(0, 1);
System.out.print(“Visits: “);
Thegraph.dfs();
System.out.println();
```

Graph

```
class graph
{
    private final int size = 20;
    private int[] st;
    private int top;

    public graph()
    {
        st = new int[size];
        top = -1;
    }

    public void push (int j)
    {
        st[++top] = j;
    }
```

Graph

```
public int pop()
    {
        return st[top--];
    }

    public int peek()
    {
        return st[top];
    }

    public boolean isempty()
    {
        return (top == -1);
    }

}
```

Graph

```
class vertex
{
    public String city;
    public boolean wasvisited;

    public vertex(String cty)
    {
        city = cty;
        wasvisited = false;
    }
}
```

Graph

```
class graphs
{
    private final int max_verts = 20;
    private vertex vertexlist[];
    private int adjmat[][];
    private int nverts;
    private graph thestack;

    public graphs()
    {
        vertexlist = new vertex[max_verts];
        adjmat = new int[max_verts][max_verts];
        nverts = 0;
        for(int j=0; j<max_verts; j++)
            for(int k= 0; k<max_verts; k++)
                adjmat[j][k] = 0;
        thestack = new graph();
    }
```

Graph

```
public void addvertex(String cty)
    {
        vertexlist[nverts++] = new vertex(cty);
    }

    public void addedge(int start, int end)
    {
        adjmat[start][end] = 1;
        adjmat[end][start] = 1;
    }

    public void displayvertex(int v)
    {
        System.out.print(vertexlist[v].city);
        System.out.print("-->");
    }
```

Graph

```
public void dfs()
    {
        vertexlist[0].wasvisited = true;
        System.out.println("From" + " " +vertexlist[0].city + "you can reach to the following cities: ");
        thestack.push(0);

        while ( !thestack.isempty() )
        {
            int v = getadjunvisitedvertex( thestack.peek() );
            if (v == -1)
                thestack.pop();
            else
            {
                vertexlist[v].wasvisited = true;
                displayvertex(v);
                thestack.push(v);
            }
        }
for(int j=0; j<nverts; j++)
            vertexlist[j].wasvisited = false;
    }
```

Graph

```
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;
    }
}
```

Graph

```
class dfsapp
{
    public static void main (String[] args)
    {
        graphs thegraph = new graphs();
        thegraph.addvertex("Kabul");
        thegraph.addvertex("Ghazni");
        thegraph.addvertex("Jalal Abad");
        thegraph.addvertex("Mazar");
        thegraph.addvertex("Qundoz");

        thegraph.addedge(0, 1);
        thegraph.addedge(1, 2);
        thegraph.addedge(0, 3);
        thegraph.addedge(3, 4);

        System.out.print("Visits: ");
        thegraph.dfs();
        System.out.println();
    }
}
```

Graph

- To use the DFS algorithm for directed graph, we need the following modification:

```
thegraph.addedge(0, 1);
thegraph.addedge(1, 0);
thegraph.addedge(0, 4);
thegraph.addedge(4, 0);
thegraph.addedge(1, 2);
thegraph.addedge(2, 1);
thegraph.addedge(2, 3);
thegraph.addedge(3, 2);
thegraph.addedge(4, 5);
thegraph.addedge(5, 4);
thegraph.addedge(4, 6);
thegraph.addedge(6, 4);
```

Graph

- Then we need to modify the addedge method as follow:

```
public void addedge(int start, int end)
    {
        adjmat[start][end] = 1;
    }
```

Breadth-First Search (BFS)

Graph

- **DFS** get as **far away** from the **starting point** as quickly as possible.
- **BFS** stay as **close as possible** to the **starting point**.
- BFS **visits all** the **vertices** adjacent to the **starting vertex**.
- Use **queue** instead of a stack.
- An **example**:
- **A** is the **starting vertex**, so you **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 is **adjacent** to the current vertex, **mark** it, and insert it into the **queue**.

Graph

- **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 are done.
- Thus, you **visit all** the vertices **adjacent** to **A**, inserting each one into the **queue** as you visit it.
- 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**, and so on.

Graph

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	
Done	

- At **each moment**, the **queue contains** the **vertices** that have been **visited** but whose **neighbors have not yet been fully explored. The node are visited in the order ABCDEFGHI.**

Graph

- **Note**: the **code** is **similar** to **DFS except** for the **inclusion** of a **queue class** instead of a **stack** class and **BFS** method instead of **DFS** method.

```
public class graphqueue
{
    private final int size = 20;
    private int[] queArray;
    private int front;
    private int rear;

    public graphqueue()
    {
        queArray = new int[size];
        front = 0;
        rear = -1;
    }
```

Graph

```
public void insert(int j)  {
        if(rear == size - 1)
            rear = -1;
        queArray[++rear] = j;
    }
public int remove()  {
        int temp = queArray[front++];
        if(front == size)
            front = 0;
        return temp;
    }
public boolean isempty() {
        return (rear + 1 == front);
    }
}
```

Graph

```
class vertex
{
    public char label;
    public boolean wasvisited;

    public vertex(char lab)
    {
        label = lab;
        wasvisited = false;
    }
}
```

Graph

```
class bfsgraph {
    private final int max_verts = 20;
    private vertex vertexlist[];
    private int adjmat[][];
    private int nverts;
    private graphqueue thequeue;
    public bfsgraph()   {
        vertexlist = new vertex[max_verts];
        adjmat = new int[max_verts][max_verts];
        nverts = 0;
        for(int j=0; j<max_verts; j++)
            for(int k= 0; k<max_verts; k++)
                adjmat[j][k] = 0;
        thequeue = new graphqueue();
    }
```

Graph

```
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)
   {
       System.out.print(vertexlist[v].label);
   }
```

Graph

```
public void bfs()  {
    vertexlist[0].wasvisited = true;
    displayvertex(0);
    thequeue.insert(0);
    int v2;
    while ( !thequeue.isempty() )  {
        int v1 = thequeue.remove();
        while( (v2 = getadjunvisitedvertex(v1)) != -1) {
            vertexlist[v2].wasvisited = true;
            displayvertex(v2);
            thequeue.insert(v2);
        }
    }
   for(int j=0; j<nverts; j++)
        vertexlist[j].wasvisited = false;
}
```

Graph

```
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;
    }
}
```

Graph

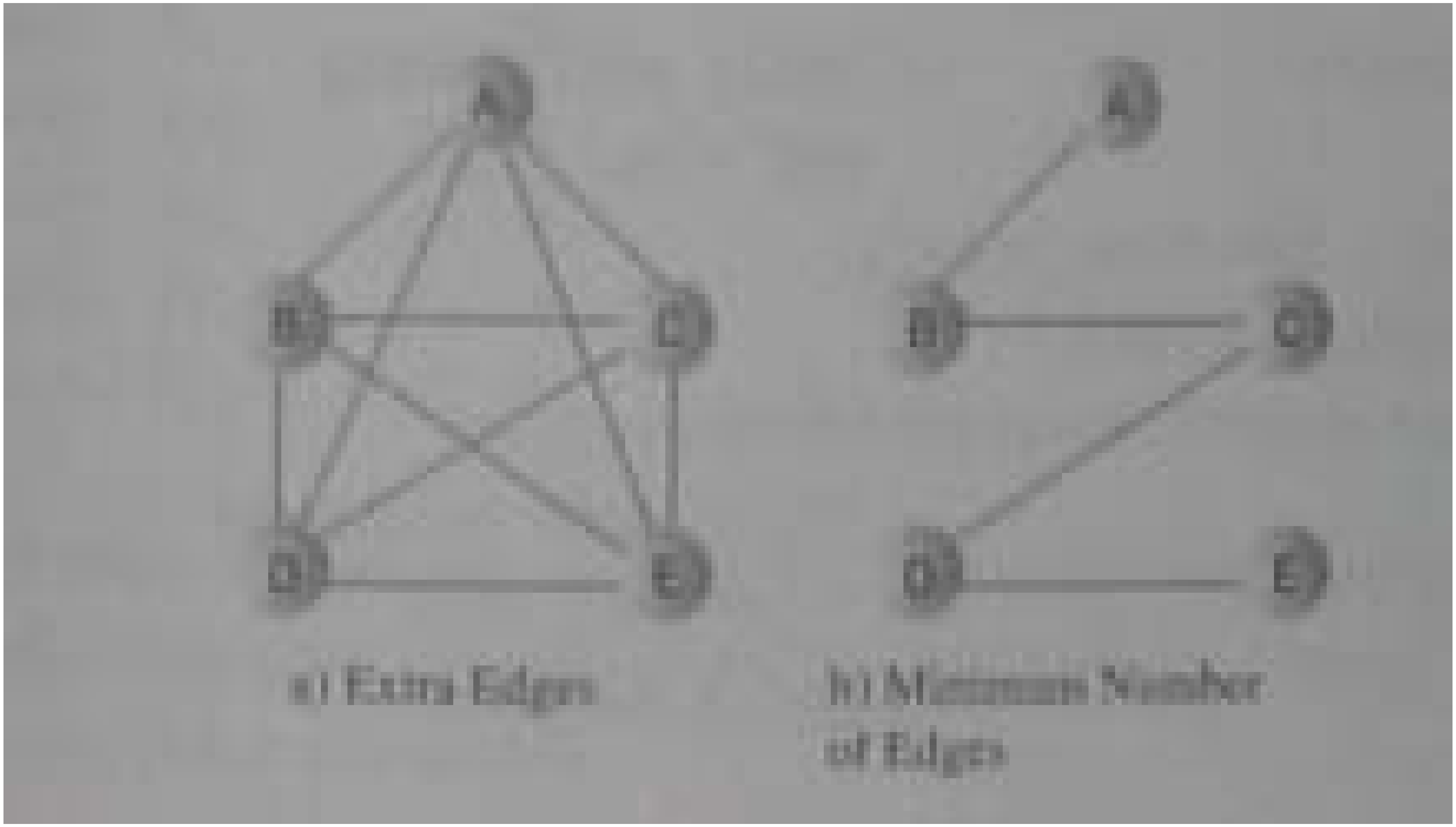
```
class bfsapp {
    public static void main(String[] args)  {
        bfsgraph thegraph = new bfsgraph();
        thegraph.addvertex('A');
        thegraph.addvertex('B');
        thegraph.addvertex('C');
        thegraph.addvertex('D');
        thegraph.addvertex('E');

        thegraph.addedge(0, 1);
        thegraph.addedge(1, 2);
        thegraph.addedge(0, 3);
        thegraph.addedge(3, 4);

        System.out.print("Visits: ");
        thegraph.bfs();
        System.out.println();
    }
}
```

Minimum Spanning Trees

Graph

- **Remove** any **extra traces**.
- The result would be a **graph** with the **minimum number** of **edges necessary** to connect the vertices.
- For **example**, **figure (a)** shows **five vertices** with an **excessive** number of **edges**, while **figure (b)** shows the **same vertices** with the **minimum** number of **edges necessary** to connect them.
- This constitutes a minimum spanning tree (**MST**).

Graph



Graph

- There are **many possible MST**.
- Figure **(b)** shows edges AB, BC, CD, and DE, but edges AC, CE, ED, and DB would do just as well.
- **E = V – 1**
- We are not **worried** here about the **length** of the **edges**. We are not trying to find a minimum **physical length**.
- The algorithm for creating the minimum spanning tree can be **based** on either the **DFS** or the **BFS**.
- By **executing** the **DFS** and **recording** the **edges** you have traveled to make the search, you automatically create a minimum spanning tree.
- The only difference between the **mst**() and **dfs**() is that **mst**() must somehow **record** the **edges traveled**.

Java Code

Graph

```
public void mst()    {
        vertexlist[0].wasvisited = true;
        thestack.push(0);
        while ( !thestack.isempty() )   {
            int currentvertex = thestack.peek();
            int v = getadjunvisitedvertex(currentvertex);
            if(v == -1)
                thestack.pop();
            else  {
                vertexlist[v].wasvisited = true;
                thestack.push(v);
```

Graph

```
                displayvertex(currentvertex);
                 displayvertex(v);
                 System.out.print("  ");
             }
         }
         for(int j=0; j<nverts; j++)
             vertexlist[j].wasvisited = false;
     }
```

Graph

```
class mststack  {
    private final int size = 20;
    private int[] st;
    private int top;
    public mststack() {
        st = new int[size];
        top = -1;
    }
    public void push (int j) {
        st[++top] = j;
    }
```

```
public int pop() {
        return st[top--];
}
public int peek() {
        return st[top];
}
public boolean isempty() {
        return (top == -1);
}
}
```

Graph

```
class vertex
{
    public char label;
    public boolean wasvisited;

    public vertex(char lab)
    {
        label = lab;
        wasvisited = false;
    }
}
```

Graph

```
class mstgraph {
   private final int max_verts = 20;
   private vertex vertexlist[];
   private int adjmat[][];
   private int nverts;
   private mststack thestack;
   public mstgraph() {
      vertexlist = new vertex[max_verts];
      adjmat = new int[max_verts][max_verts];
      nverts = 0;
      for(int j=0; j<max_verts; j++)
         for(int k= 0; k<max_verts; k++)
            adjmat[j][k] = 0;
      thestack = new mststack();
   }
```

Graph

```
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)
    {
        System.out.print(vertexlist[v].label);
    }
```

Graph

```
public void mst()    {
        vertexlist[0].wasvisited = true;
        thestack.push(0);
        while ( !thestack.isempty() )    {
            int currentvertex = thestack.peek();
            int v = getadjunvisitedvertex(currentvertex);
            if(v == -1)
                thestack.pop();
            else  {
                vertexlist[v].wasvisited = true;
                thestack.push(v);

                displayvertex(currentvertex);
                displayvertex(v);
                System.out.print("  ");
            }
        }
        for(int j=0; j<nverts; j++)
            vertexlist[j].wasvisited = false;
    }
```

Graph

```
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;
  }
}
```

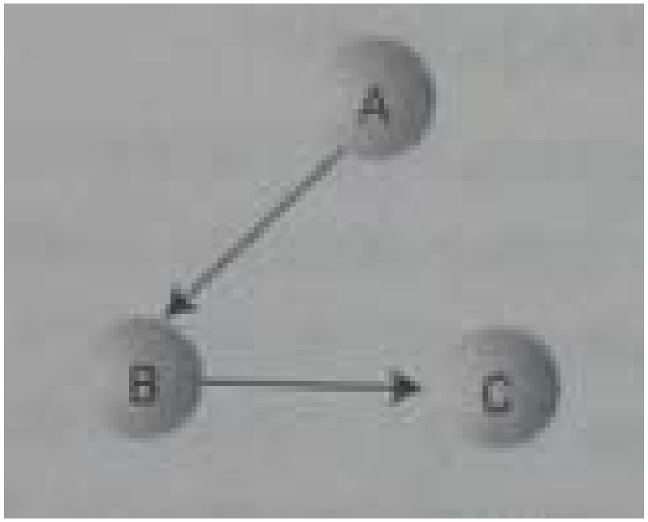
Graph

```
class mstapp  {
   public static void main (String[] args)  {
      mstgraph thegraph = new mstgraph();
      thegraph.addvertex('A');
      thegraph.addvertex('B');
      thegraph.addvertex('C');
      thegraph.addvertex('D');
      thegraph.addvertex('E');
      thegraph.addedge(0, 1);
      thegraph.addedge(0, 2);
      thegraph.addedge(0, 3);
      thegraph.addedge(0, 4);
      thegraph.addedge(1, 2);
      thegraph.addedge(1, 3);
      thegraph.addedge(1, 4);
      thegraph.addedge(2, 3);
      thegraph.addedge(2, 4);
      thegraph.addedge(3, 4);
      System.out.print("Minimum Spanning Tree:  ");
      thegraph.mst();
      System.out.println();
   }
}
```

Directed Graph

Graph

- The graph needs a **feature**: The **edge** need to have a **direction**. When this is the case, the graph is called a **directed graph**. In a directed graph you can **proceed only one way** along an edge. The arrows in the figure show the **direction** of the **edges**.



Graph

- 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. The following figure shows the adjacency matrix for the above figure:

	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**.
- 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**.

Graph

- 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;
}
```

- **Connectivity in Directed Graphs**
- We have seen how in a **non-directed** graph you can **find all** the **vertices** that are **connected** by **doing** a **depth-first** or **breadth-first search**. When we try to find all the **connected vertices** in a **directed** graph, things get more **complicated**. You can't just start from a randomly selected vertex and expect to reach all the other connected vertices. Consider the following graph:
- If you **start on A,** you can **get** to **C** but not to any of the other vertices. If you **start on B**, you **can't** get to **D,** and if you start **on C**, you can't get **anywhere**. The **meaningful question** about connectivity is: **What vertices** can you reach if you start on a **particular vertex**?

Warshall's Algorithm

Graph

- In **some application** it is important to find out **quickly** whether one vertex is **reachable** from **another vertex**.
- You could examine the **connectivity table**, but then you would need to look through **all** the **entries** on a **given row**, which would take **O(N) time. But** you are in a **hurry**; is there a **faster way**?
- It is possible to **construct** a **table** that will tell you **instantly** (that is, O(1) time) whether **one vertex** is **reachable from another**. Such a table can be **obtained** by **systematically** modifying a graph's **adjacency matrix**. The **graph represented** by this **revised adjacency** matrix is called the **transitive closure** of the **original graph**.
- In an **ordinary adjacency matrix** the **row number indicates** where an edge **starts** and the **column** number indicates where it **ends**. A **1** at the **intersection** of **row C** and **column D** means there is an **edge** from vertex **C** to vertex **D**. You can get from one vertex to the other in one step.

Graph

	A	B	C	D	E
A	0	0	1	0	0
B	1	0	0	0	1
C	0	0	0	0	0
D	0	0	0	0	1
E	0	0	1	0	0

Graph

- We can **use Warshall's algorithm** to **change** the **adjacency matrix** into the **transitive closure** of the graph. This algorithm does a **lot** in a few **lines** of **code**. It is based on a **simple idea**:
- If **you can get** from **vertex L** to vertex **M,** and you can get **from M to N**, then you can get from **L to N**.
- We have **derived** a **two-step path** from two **one-step paths**. The **adjacency matrix** shows **all possible one-step** paths, so it is a good **starting place** to apply this rule.
- **Row A**
- We **start** with **row A**. There is **nothing** in **columns A** and **B**, but there is a **1** at **column C**, so we **stop there**. Now the **1 at this location** says there is a path from **A to C**. If we knew there was a path from some vertex **X to A,** then we would know there was a path from **X to C**. Where are the edges (**if any**) that **end** at **A**? They are in **column A**. So we examine all the **cells** in **column A**. In the above table there is only one **1** in **column A**: at **row B**. It says there is an **edge** from **B to A**. So we know there is an edge from **B to A**, and another (the one we started with) from **A** to **C.** From this we infer that we can get from **B to C** in two steps. You can verify this is true by looking at the graph.
- To **record** this **result**, we put a **1** at the intersection of row **B and column C**. The result is shown in the **following table**:
- The **remaining cells** of **row A are blank**.

Graph

- **Rows B, C, and D**
- We go to **row B**. The **first cell**, at **column A** has a **1,** indicating an **edge** from **B to A**. Are there **any edges** that **end** at **B**? We **look** in **column B**, but it is **empty**, so we know that none of the **1s** we find in **row B** will result in finding **longer paths** because **no edges** end at **B**.
- **Row C** has no **1s** at all, so we go to **row D**. Here we find an edge from **D to E**. However, **column D** is **empty**, so there are no edges that end on D.
- **Row E**
- In **row E** we see there is an **edge** from **E to C.** Looking in **column E** we see the first entry is for the edge **B to E**, so with **B to E** and **E to C** we infer there is a path from **B to C**. However, it is already been discussed, as indicated by the 1 at that location.
- There is **another 1** in **column E,** at **row D.** This edge **from D to E** plus the one from **E to C imply** a **path** from **D to C,** so we **insert** a **1 in** that **cell**.
- **Warshall's algorithm** is now **complete**.

Implementation of Warshall Algorithm

Graph

- One way to **implement** Warshall algorithm is with **3 nested loops**. The **outer loop** looks at **each row**; let's call its **variable y.** The loop **inside** that looks at each **cell** in the **row**; it use **variable x.** If a **1** is **found** in cell (**x,y**), there is an **edge** from **y to x**, and the **third** (innermost) **loop** is activated; it use **variable z**. The **third loop examines** the **cells** in **column y**, looking for an **edge** that **ends** at **y**. (Note that y is used for rows in the first loop but for the column in the third loop). If there is a **1** in **column y** at **row z**, then there is an edge from **z to y**. with one **edge** from **z to y** and another from **y to x** it follows that there is a **path** from **z to x,** so you can put a **1** at (**x,z**).

Weighted Graphs

Graph

- If **vertices** in a **weighted graph** represent **cities**, the weight of the edges might represent **distance between** the **cities**, or **costs** to **fly** between them. When we include weight as a **feature** of a graph's **edges**, some interesting and complex questions arise. What is the **minimum spanning tree** for a **weighted graph**?
- What is the **shortest** (or **cheapest**) **distance** from one **vertex** to **another**?
- Such **questions** have **important applications** in the **real world**.

Minimum Spanning Tree with Weighted Graphs

Graph

- To introduce weighted graphs, we will return to the question of the **minimum spanning tree**. Creating such a tree is a bit more **complicated** with a **weighted graph** than with an unweighted one. When **all edges** are the same **weight**, it is fairly **straightforward** for the **algorithm** to choose one to add to the **minimum spanning tree**. But when edges can have **different weights**, some **arithmetic** is needed to choose the right one.

Creating the Algorithm

Graph

- The **key activity** in carrying out the algorithm is to **maintain** a **list** of the **costs** of links between pairs of **nodes**. A list in which we **repeatedly** select the **minimum** value suggests a **priority queue** as an **appropriate data structure**. In a serious program this priority queue might be based on a **heap** and this would speed up operations on large priority queues. However, in our example we will use a **simple array**.
- **Outline of the Algorithm**
- **Start** with a **vertex**, and **put** it in the **tree**. Then **repeatedly** do the **following**:
- **Find** all the **edges** from the **newest vertex** to other **vertices** that are not in the **tree**. **Put** these **edges** in the **priority queue**.
- **Pick** the **edge** with the **lowest weight**, and **add** this **edge** and its **destination vertex** to the **tree**.
- **Repeat** these **steps** until all the **vertices** are in the tree. At that point you are done.

Graph

- In a **programming algorithm** we make **sure** that we do not have **any edges** in the **priority queue** that **lead** to **vertices** that are already in the tree. We could go **through** the **queue looking** for and **removing** any such **edges each time** we **added** a new **vertex** to the **tree**. As it turns out, it is **easier** to **keep only one edge from** the **tree** to a **given vertex** in the **priority queue** at any given time.

Graph

- **Java code:**
- The **following method** creates the **minimum spanning tree** for a **weighted graph**, follows the algorithm outlined earlier. As in our other graph programs, it assumes there is a list of vertices in **vertexlist**[], and that it will start with the **vertex** at **index 0**. the **currentvertex variable** represents the **vertex most recently** added to the **tree**.

Graph

```
public void mstw() {
    currentvertex = 0;
    while(ntree < nverts – 1) {
        vertexlist[currentvertex] .isintree = true;
        ntree++;
        // insert edges adjacent to currentvertex into PQ
        for(int j = 0;  j<nverts; j++)
        {
            if(j == currentvertex)
                continue;
            if(vertexlist[j].isintree)
                continue;
            int distance = adjmat[currentvertex][j];
            if(distance == infinity)
                continue;
            putinpq(j, distance);
        }
```

Graph

```
    if(thepq.size() == 0)
        {
                System.out.println(“Graph not connected”);
                return;
        }
        // remove edge with minimum distance, from pq
        edge theedge = thepq.removemin();
        int sourcevert = the edge.srcvert;
        currentvert = theedge.destvert;
        // display edge from source to current
        System.out.print( vertexlist[sourcevert].abel);
        System.out.print(vertexlist[currentvertex].label);
        System.out.print(“   “);
    }

    for(int j=0; j<nverts; j++)
        vertexlist[j].isintree = false;
}
```

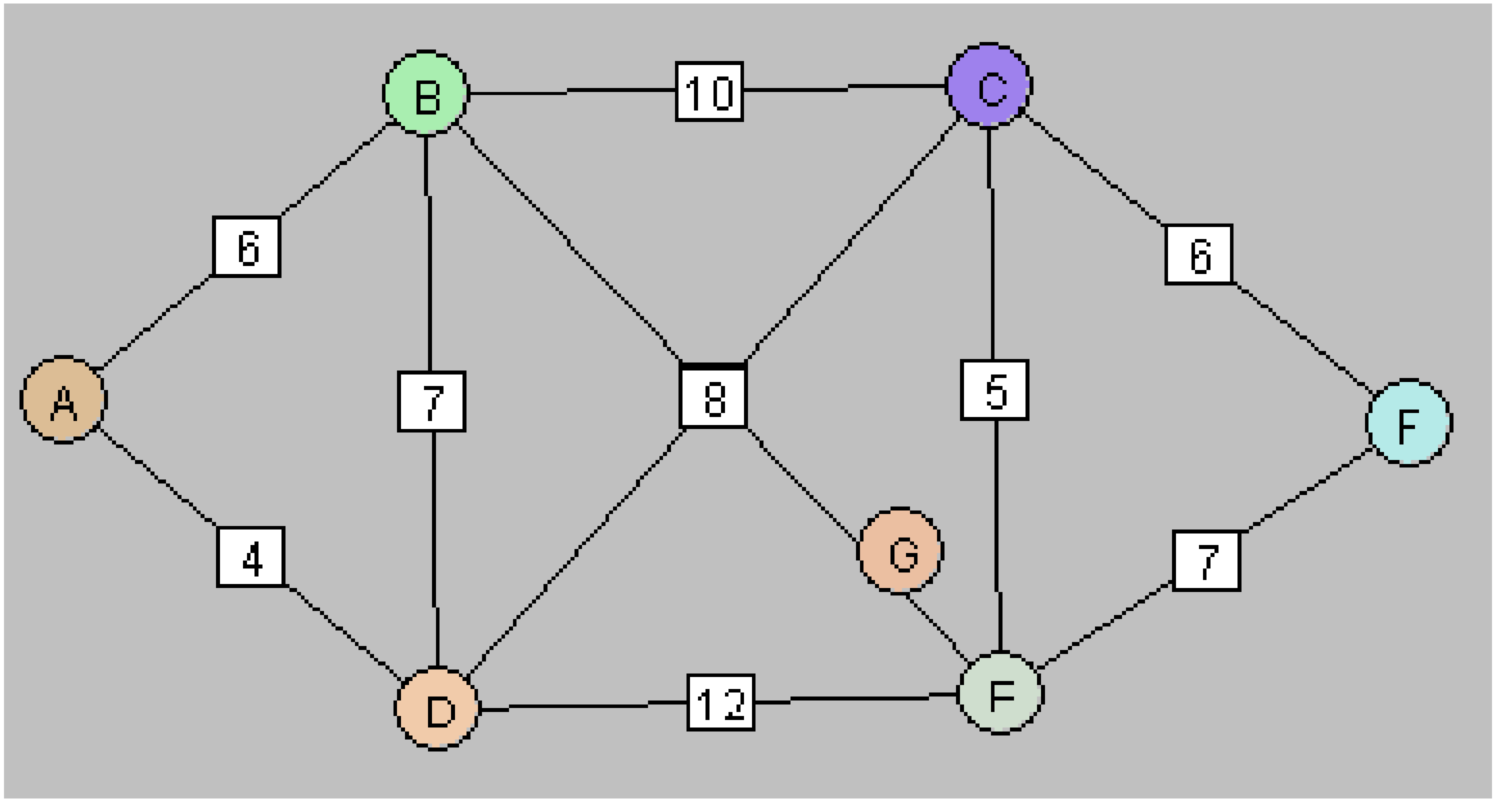
Graph

- The algorithm is carried out in the while loop, which terminate when all vertices are in the tree. Within this loop the following activities take place:
- The current vertex is placed in the tree.
- The edges adjacent to this vertex are placed in the priority queue.
- Edge with the minimum weight is removed from the priority queue. The destination vertex of this edge becomes the current vertex.
- The following is the code for putinpq() method:

Graph

```
public void putinpq(int newvert, int newdist) {
    int queueindex = thepq.find(newvert);
    if(queueindex != -1) {
        edge tempedge = thepq.peekn(queueindex);
        int olddist = tempedge.distance;
        if(olddist > newdist)
        {
            thepq.removen(queueindex);
            edge theedge = new edge(currentvertex, newvert, newdist);
            thepq.insert(theedge);
        }
    }
    else
    {
        edge theedge = new edge(currentvertex, newvert, newdist);
        thepq.insert(theedge);
    }
}
```

Graph



Graph

```
class edge
{
    public int srcvert;
    public int destvert;
    public int distance;

    public edge(int sv, int dv, int d)
    {
        srcvert = sv;
        destvert = dv;
        distance = d;
    }
}
```

Graph

```
class graph {
    private final int max_verts = 20;
    private final int infinity = 10000;
    private vertex vertexlist[];
    private int adjmat[][];
    private int nverts;
    private int currentvert;
    private priorityq thepq;
    private int ntree;
    public graph()  {
        vertexlist = new vertex[max_verts];
        adjmat = new int[max_verts][max_verts];
        nverts = 0;
        for(int j=0; j<max_verts; j++)
            for(int k=0; k<max_verts; k++)
                adjmat[j][k] = infinity;
        thepq = new priorityq();
    }
```

Graph

```
public void addvertex(char lab)
{
    vertexlist[nverts++] = new vertex(lab);
}

public void addedge(int start, int end, int weight)
{
    adjmat[start][end] = weight;
    adjmat[end][start] = weight;
}

public void displayvertex(int v)
{
    System.out.print(vertexlist[v].label);
}
```

Graph

```
public void mstw()
{
    currentvert = 0;
    while(ntree < nverts-1)
    {
        vertexlist[currentvert].isintree = true;
        ntree++;
        for(int j=0; j< nverts; j++)
        {
            if(j==currentvert)
                continue;
            if(vertexlist[j].isintree)
                continue;
            int distance = adjmat[currentvert][j];
            if(distance == infinity)
                continue;
            putinpq(j, distance);
        }
```

Graph

```
if(thepq.size2() == 0)
         {
            System.out.println("Graph Not Connected");
            return;
         }

         edge theedge = thepq.removemin();
         int sourcevert = theedge.srcvert;
         currentvert = theedge.destvert;

         System.out.print(vertexlist[sourcevert].label);
         System.out.print(vertexlist[currentvert].label);
         System.out.print("  ");
      }

      for(int j=0; j<nverts; j++)
         vertexlist[j].isintree = false;
   }
```

Graph

```
public void putinpq(int newvert, int newdist) {
        int queindex = thepq.find(newvert);
        if(queindex != -1)
        {
            edge tempedge = thepq.peekn(queindex);
            int olddist = tempedge.distance;
            if(olddist > newdist)
            {
                thepq.removen(queindex);
                edge theedge = new edge(currentvert, newvert, newdist);
                thepq.insert(theedge);
            }
        }
        else
        {
            edge theedge = new edge(currentvert, newvert, newdist);
            thepq.insert(theedge);
        }
    }
}
```

Graph

```
class vertex
{
    public char label;
    public boolean isintree;

    public vertex(char lab)
    {
        label = lab;
        isintree = false;
    }


}
```

Graph

```
class priorityq {
    private final int size = 20;
    private edge[] quearray;
    private int size2;
    public priorityq() {
        quearray = new edge[size];
        size2 = 0;
    }
     public void insert(edge item) {
        int j;
        for(j=0; j<size2; j++)
            if(item.distance >= quearray[j].distance)
                break;
        for(int k=size2-1; k>=j; k--)
            quearray[k+1] = quearray[k];
        quearray[j] = item;
        size2++;
    }
```

Graph

```
public edge removemin() {
        return quearray[--size2];
    }
    public void removen(int n) {
        for(int j=n; j<size2-1; j++)
            quearray[j] = quearray[j+1];
        size2--;
    }
   public edge peekmin() {
        return quearray[size2-1];
    }
   public int size2() {
        return size2;
    }
```

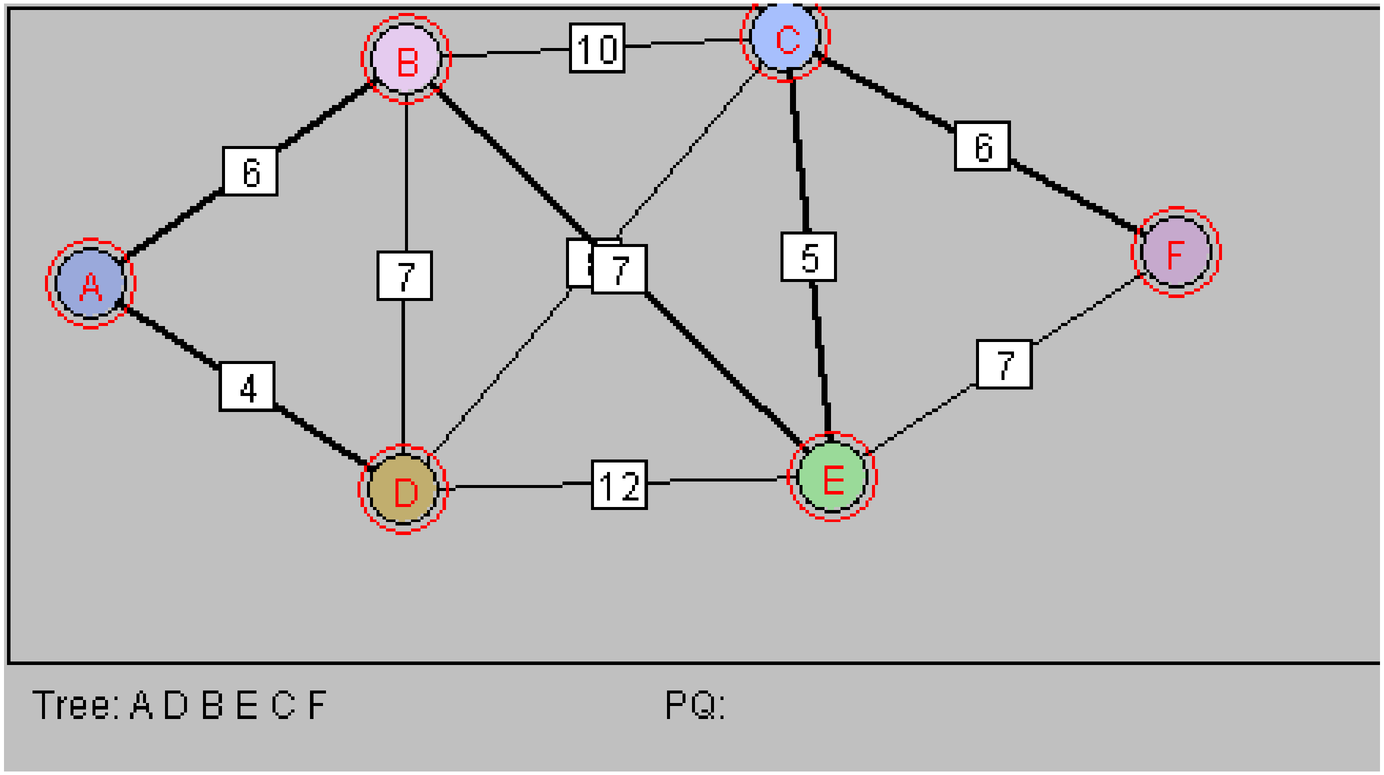
Graph

```
public boolean isempty()
    {
        return (size2 == 0);
    }
    public edge peekn(int n)
    {
        return quearray[n];
    }
   public int find(int finddex)
    {
        for (int j=0; j<size2; j++)
            if(quearray[j].destvert == finddex)
                return j;
        return -1;
    }
}
```

Graph

```
class mstwapp {
    public static void main(String[] args) {
        graph thegraph = new graph();
        thegraph.addvertex('A');
        thegraph.addvertex('B');
        thegraph.addvertex('C');
        thegraph.addvertex('D');
        thegraph.addvertex('E');
        thegraph.addvertex('F');
        thegraph.addedge(0,1,6);
        thegraph.addedge(0,3,4);
        thegraph.addedge(1,2,10);
        thegraph.addedge(1,3,7);
        thegraph.addedge(1,4,7);
        thegraph.addedge(2,3,8);
        thegraph.addedge(2,4,5);
        thegraph.addedge(2,5,6);
        thegraph.addedge(3,4,12);
        thegraph.addedge(4,5,7);
        System.out.print("Minimum Spanning Tree: ");
        thegraph.mstw();
        System.out.println();
    }
}
```

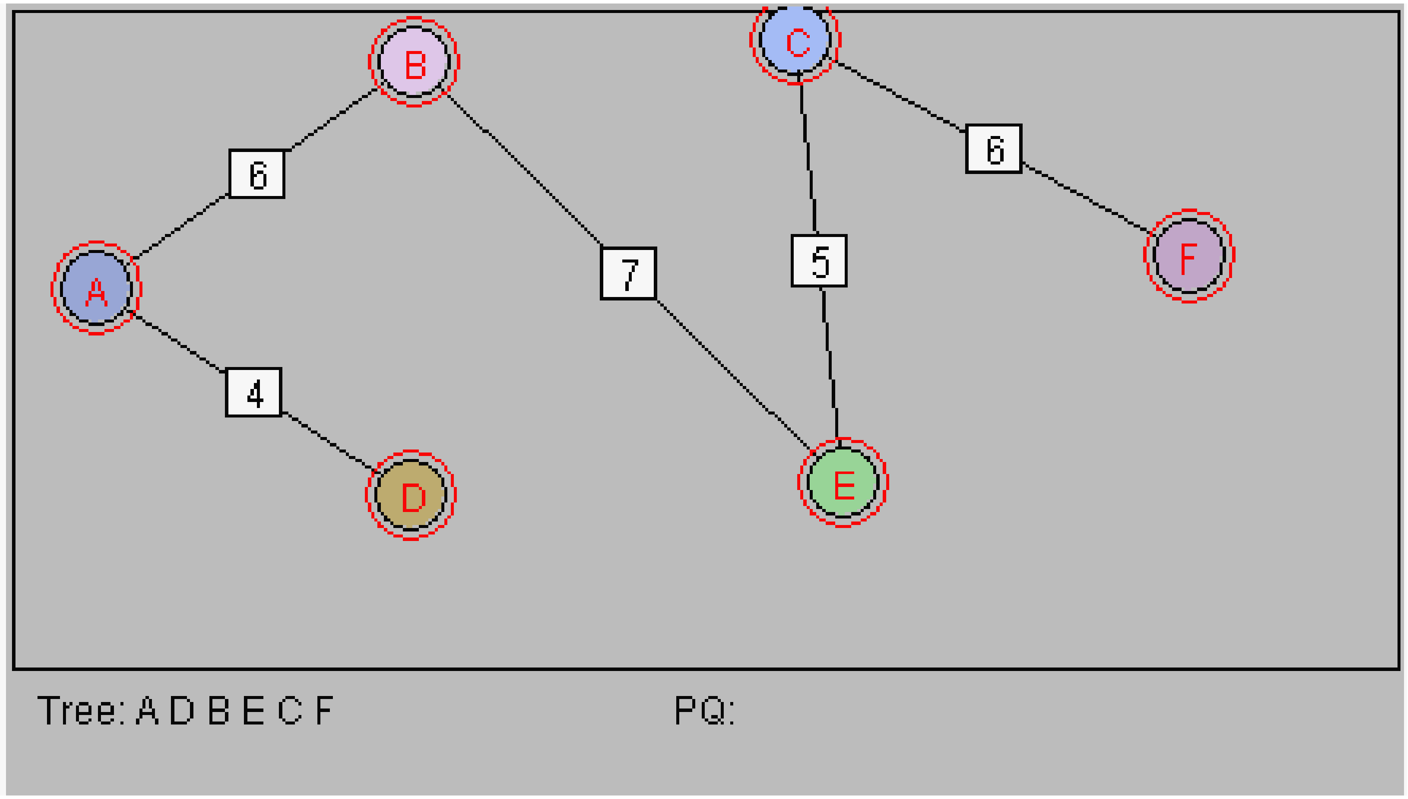
Graph



Tree: A D B E C F

PQ:

Graph



Tree: A D B E C F

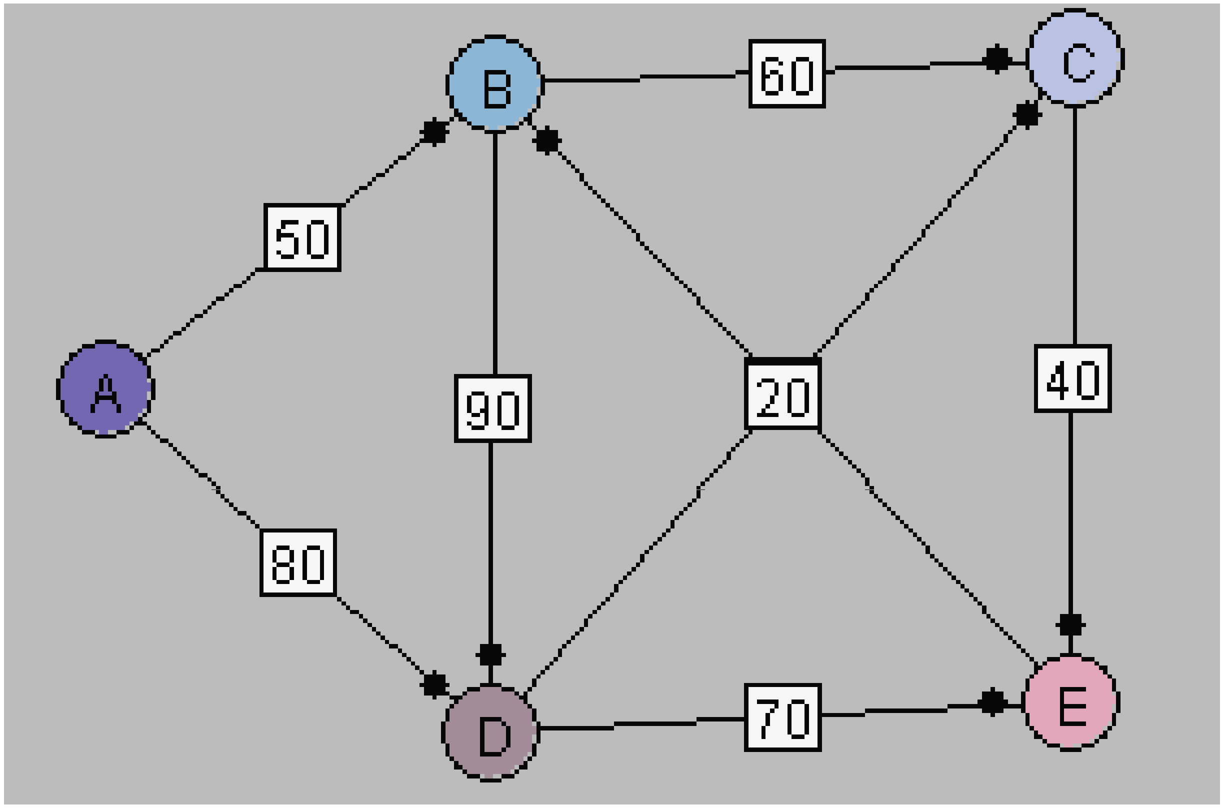
PQ:

The Shortest-Path Problem

Graph

- The **most commonly encountered** problem associated with **weighted graphs** is that of finding the **shortest path between** two given **vertices**. This solution to this problem is applicable to a wide variety of **real-world situations**. It is more **complex problem** than we have seen before.
- The **shortest-path problem** is this: for a given **starting point** and destination, what is the **cheapest route**?
- **Dijkstra's Algorithm**
- The solution for the **shortest-path** problem is called **Dijkstar's algorithm** after **Edsger Dijkstra**, who first described it in **1959**. This algorithm is based on the **adjacency matrix** representation of a graph. This algorithm finds not only the shortest path from one specified vertex to another, but also the **shortest paths** from the **specified vertex** to **all** the other **vertices**.

Graph



Java code

Graph

- The **code** for the shortest-path algorithm may be the **most complex**. The **key data structure** in the shortest-path algorithm is an **array** that **keeps** track of the **minimum distances** from the **starting vertex** to the other **vertices** (destination vertices). During the execution of the algorithm, these **distances** are **changed**, until at the **end** they hold the **actual shortest distances** from the **start**. In the example code, this array is called **spath**[].
- It is **important** to **record not only** the **minimum distance** from the **starting vertex** to each destination vertex, but also the **path taken**. Fortunately, the entire path need not be **explicitly stored**. It is only necessary to store the **parent** of the **destination vertex**. The **parent** is the vertex **reached** just before the **destination**. There are **several ways** to keep track of the **parent vertex**, but we choose to combine the parent with the distance and put the resulting object into the **spath**[] array. We call this class of objects **Distpar** (for distance parent).

Graph

```
class distpar
{
   public int distance;
   public int parentvert;
public distpar (int pv, int d)
{
   distance = d;
   parentvert = pv;
}
}
```

Graph

- **The path() method:**
- The **path()** method carries out the **actual shortest-path algorithm**. It uses the distpar **class** and the **vertex class**, which we saw in the **mstw**. The **path**() method is a member of the **graph class**.

Graph

```
public void path() {
        int starttree =0;
        vertexlist[starttree].isintree = true;
        ntree = 1;
        for(int j=0; j<nverts; j++)
        {
            int tempdist = adjmat[starttree][j];
            spath[j] = new distpar(starttree, tempdist);
        }
        while(ntree < nverts)
        {
            int indexmin = getmin();
            int mindist = spath[indexmin].distance;
```

Graph

```
if(mindist == infinity) {
                System.out.println("There are unreachable vertices");
                break;
            }
            else
            {
                currentvert = indexmin;
                starttocurrent = spath[indexmin].distance;
            }
            vertexlist[currentvert].isintree =true;
            ntree++;
            adjust_spath();
        }
        displaypaths();

        ntree =0;
        for(int j=0; j<nverts; j++)
            vertexlist[j].isintree = false;
    }
```

Graph

- The **starting** vertex is always at **index 0** of the **vertexlist**[ ] array. The **first task** in **path**() is to **put** this **vertex** into the **tree**. As the algorithm proceeds, we will be **moving** other **vertices** into the tree as well. The **vertex object** contain a flag that indicates whether a **vertex object** is in the tree. Putting a vertex in the tree consists of setting this flag and incrementing **ntree**, which counts how many **vertices** are in the tree.
- Second, **path**() copies the distance from the appropriate row of the adjacency matrix to **spath**[ ]. This is always **row 0**, because for simplicity we assume **0** is the **index** of the **starting vertex**. Initially, the parent field of all the **spath**[ ] entries is A, the **starting vertex**.
- The **while loop** of the algorithm terminates after all the vertices have been placed in the **tree**.

Graph

- **There** are **basically 3 actions** in this **loop**:
- **Choose** the **spath**[ ] entry with the **minimum distance**.
- **Put** the **corresponding vertex** in the **tree**. This becomes the "**current vertex**" **currentvert**.
- **Update all** the **spath** [ ] entries to **reflect distances** from **currentvert**.
- If **path**() finds that the **minimum distance** is **infinity**, it knows that some vertices are **unreachable** from the **starting point**. Why? Because not all the **vertices** are in the tree (the while loop has not **terminated**), and yet there is no way to get to these extra **vertices**; if there were, there would be a **non-infinite** distance.
- To find the **spath**[ ] entry with the **minimum distance**, **path**() calls the **getmin**() method. This **straightforward**; it steps across the spath[ ] entries and return with the column number of the entry with the **minimum distance**.

Graph

- **Updating spath[ ] with adjust_spath():**
- The **adjust_spath**() method is used to **update** the **spath**[ ] entries to **reflect new information** obtained from the **vertex** just **inserted** in the tree. When this routine is called, **currentvert** has just been placed in the **tree**, and **starttocurrent** is the current entry in **spath** [ ] for this **vertex**. The **adjust_spath**() method now **examine** each **vertex** entry in **spath**[ ], using the **loop counter column** to point to each vertex in turn.
- For each **spath**[ ] entry, provided the **vertex** is not in the tree, it does **three things**:
- It **adds** the **distance** to the **current** (already calculated and now in starttocurrent) to the **edge distance** from **currentvert** to the column **vertex. we** call the result **starttofringe**.
- It **compares starttofringe** with the **current** entry in **spath** [ ].
- If **starttofringe** is less, it replaces the entry in **spath** [ ].
- This is the **heart** of **Dijkstra's algorithm**. It keeps **spath** [ ] **updated** with the shortest **distances** to all the vertices that are **currently** known.

Java program for Dijkstra’s Algorithm:

```
class distpar
{
    public int distance;
    public int parentvert;

    public distpar(int pv, int d)
    {
        distance = d;
        parentvert = pv;
    }
}
```

Graph

```
class vertex2
{
    public char label;
    public boolean isintree;

    public vertex2(char lab)
    {
        label = lab;
        isintree = false;
    }

}
```

Graph

```
class graph2
{
    private final int max_verts = 20;
    private final int infinity = 10000;
    private vertex2 vertexlist[];
    private int adjmat[][];
    private int nverts;
    private int ntree;
    private distpar spath[];
    private int currentvert;
    private int starttocurrent;
```

Graph

```
public graph2() {
        vertexlist = new vertex2[max_verts];
        adjmat = new int[max_verts][max_verts];
        nverts = 0;
        ntree = 0;
        for(int j=0; j<max_verts; j++)
            for(int k=0; k<max_verts; k++)
                adjmat[j][k] = infinity;
        spath = new distpar[max_verts];
    }
    public void addvertex(char lab)
    {
        vertexlist[nverts++] = new vertex2(lab);
    }
   public void addedge(int start, int end, int weight)
    {
        adjmat[start][end] = weight;
    }
```

Graph

```
public void path() {
        int starttree =0;
        vertexlist[starttree].isintree = true;
        ntree = 1;
        for(int j=0; j<nverts; j++)
        {
            int tempdist = adjmat[starttree][j];
            spath[j] = new distpar(starttree, tempdist);
        }
        while(ntree < nverts) {
            int indexmin = getmin();
            int mindist = spath[indexmin].distance;
            if(mindist == infinity)
            {
           System.out.println("There are unreachable
      vertices");
                break;
            }
```

```
Else {
       currentvert = indexmin;
       starttocurrent =
spath[indexmin].distance;
            }
       vertexlist[currentvert].isintree =true;
            ntree++;
            adjust_spath();
        }
        displaypaths();
        ntree =0;
        for(int j=0; j<nverts; j++)
            vertexlist[j].isintree = false;
    }
```

Graph

```
public int getmin()
   {
      int mindist = infinity;
      int indexmin = 0;
      for(int j=1; j<nverts; j++)
      {
         if(!vertexlist[j].isintree && spath[j].distance < mindist)
         {
            mindist = spath[j].distance;
            indexmin = j;
         }
      }
      return indexmin;
   }
```

Graph

```
public void adjust_spath() {
        int column = 1;
        while(column < nverts) {
            if(vertexlist[column].isintree)
            {
                column++;
                continue;
            }
            int currenttofringe = adjmat[currentvert][column];
            int starttofringe = starttocurrent + currenttofringe;
            int spathdist = spath[column].distance;
            if(starttofringe < spathdist)
            {
                spath[column].parentvert = currentvert;
                spath[column].distance = starttofringe;
            }
            column++;
        }
    }
```

Graph

```
public void displaypaths()
    {
        for(int j=0; j<nverts; j++)
        {
            System.out.print(vertexlist[j].label + "=");
            if(spath[j].distance == infinity)
                System.out.print("inf");
            else
                System.out.print(spath[j].distance);
            char parent = vertexlist[spath[j].parentvert].label;
            System.out.print("(" + parent + " ) ");
        }
        System.out.println(" ");
    }
}
```

Graph

```
class pathapp {
   public static void main (String[] args)  {
      graph2 thegraph = new graph2();
      thegraph.addvertex('A');
      thegraph.addvertex('C');
      thegraph.addvertex('B');
      thegraph.addvertex('D');
      thegraph.addvertex('E');
      thegraph.addedge(0,1,50);
      thegraph.addedge(0,3,80);
      thegraph.addedge(1,2,60);
      thegraph.addedge(1,3,90);
      thegraph.addedge(2,4,40);
      thegraph.addedge(3,2,20);
      thegraph.addedge(3,4,70);
      thegraph.addedge(4,1,50);
      System.out.println("Shotest paths");
      thegraph.path();
      System.out.println();
   }
}
```

Graph