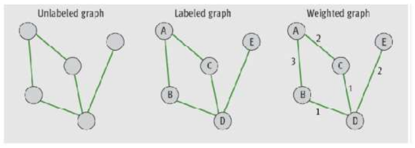
Graphs

Introduction

- Mathematically, a graph is a set V of vertices and a set E of edges, such that each edge in E connects two of the vertices in V
- We use node as a synonym for vertex

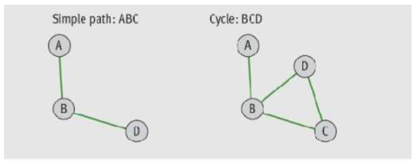


Terminology

- One vertex is **adjacent** to another vertex if there is an edge connecting the two vertices (neighbors)
- **Path**: Sequence of edges that allows one vertex to be reached from another vertex in a graph
- **Length of a path**: Number of edges on the path
- **Degree of a vertex**: Number of edges connected to it

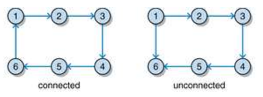
Terminology

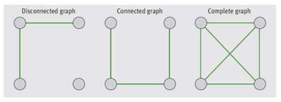
- **Simple path**: Path that does not pass through the same vertex more than once
- **Cycle**: Path that begins and ends at the same vertex



Terminology

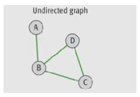
- A graph is **connected** if there is a path from each vertex to every other vertex
- A graph is **complete** if there is an edge from each vertex to every other vertex

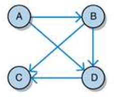


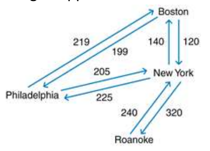


Graph Terminology

- Graphs can be undirected or directed (digraph)
- A directed edge has a source vertex and a destination vertex
- Edges emanating from a given source vertex are called its incident edges
- To convert undirected graph to equivalent directed graph, replace each edge in undirected graph with a pair of edges pointing in opposite directions

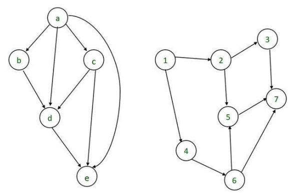




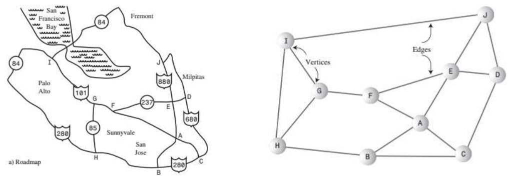


Terminology

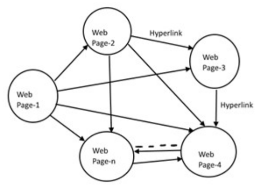
- A special case of digraph that contains no cycles is known as a **directed acyclic graph**, or **DAG**



Graph Applications – Example - Roadmap



Graph Applications – Example – Web Links



Graph Applications

- Graphs serve as models of a wide range of objects:
    - A roadmap
    - A map of airline routes
    - A schematic of the computers and connections that make up the Internet
    - The links between pages on the Web
    - The relationship between students and courses i.e courses and their pre-requisites
    - A diagram of the flow capacities in a communications or transportation network

Representing Graphs

- To represent graphs, you need a convenient way to store the vertices and the edges that connect them
- Two commonly used representations of graphs:
    - The adjacency matrix
    - The adjacency list

Adjacency Matrix

- If a graph has N vertices labeled 0, 1, . . . , N – 1:
- The adjacency matrix for the graph is a grid G with N rows and N columns
- Cell G[i][j] = 1 (or F) if there’s an edge from vertex i to j
- Otherwise, there is no edge and that cell contains 0

Adjacency Matrix – A Directed Graph

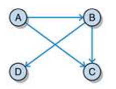
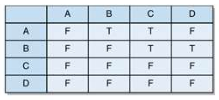


FIGURE 13.15 A directed graph



	A	B	C	D
A	F	T	T	F
B	F	F	T	T
C	F	F	F	F
D	F	F	F	F

FIGURE 13.16 An adjacency matrix for a directed graph

Cell G[i][j] = T if there's an edge from vertex i to j
Cell G[i][j] = F if there's no edge from vertex i to j

Adjacency Matrix - An Undirected Graph

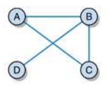
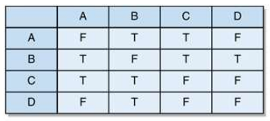


FIGURE 13.13 An undirected graph



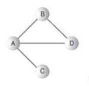
	A	B	C	D
A	F	T	T	F
B	T	F	T	T
C	T	T	F	F
D	F	T	F	F

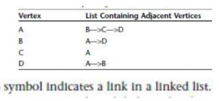
FIGURE 13.14 An adjacency matrix for an undirected graph

Cell G[i][j] = T if there's an edge from vertex i to j
Cell G[i][j] = F if there's no edge from vertex i to j

Adjacency List

- If a graph has N vertices labeled 0, 1, . . . , N – 1,
- The adjacency list for the graph is an array of N linked lists
- Each individual list shows what vertices a given vertex is adjacent to.
- Adjacency list shows which vertices are adjacent to—that is, one edge away from—a given vertex, not paths from vertex to vertex.

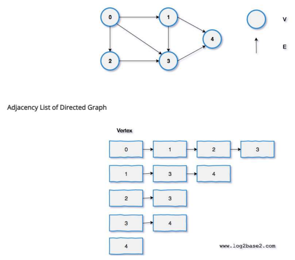




Vertex	List Containing Adjacent Vertices
A	B—>C—>D
B	A—>D
C	A
D	A—>B

—> symbol indicates a link in a linked list.

Adjacency List – Directed Graph



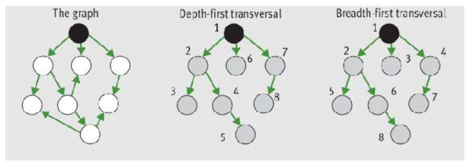
Vertex			
0	1	2	3
1	3	4	
2	3		
3	4		
4			

Graph Traversals

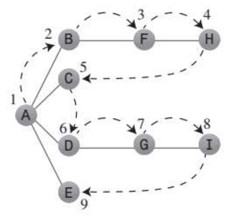
- In addition to the insertion and removal of items, important graph-processing operations include: – Traversing all of the items in a graph
- One starts at a given vertex and, from there, visits all vertices to which it connects
- Different from tree traversals, which always visit all of the nodes in a given tree
- There are two algorithms (breadth first traversal/search and depth first search/algorithm)
- Finding the shortest path to a given item in a graph

Graph Traversal

- A depth-first search (DFS), uses a stack
- A breadth-first search (BFS), uses a queue



Depth First Search



Depth First Search

- To carry out the depth-first search, you pick a starting point—in this case, vertex A.
- You then do three things: visit this vertex, push it onto a stack so you can remember it, and mark it so you won't visit it again.
- Next, you go to any vertex adjacent to A that hasn't yet been visited. We'll assume the vertices are selected in alphabetical order, so that brings up B. You visit B, mark it, and push it on the stack.
- Now what? You're at B, and you do the same thing as before: go to an adjacent vertex that hasn't been visited. This leads you to F. We can call this process Rule 1.

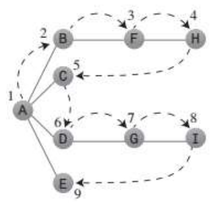
Depth First Search

- **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.

Depth First Search

- Following this rule, you pop H off the stack, which brings you back to F. F has no unvisited adjacent vertices, so you pop it. Ditto B. Now only A is left on the stack.
- A, however, does have unvisited adjacent vertices, so you visit the next one, C. But C is the end of the line again, so you pop it and you're back to A. You visit D, G, and I, and then pop them all when you reach the dead end at I. Now you're back to A. You visit E, and again you're back to A.
- This time, however, A has no unvisited neighbors, so we pop it off the stack. But now there's nothing left to pop, which brings up Rule 3.
- RULE 3: If you can't follow Rule 1 or Rule 2, you're done

Depth Fist Search



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

DFS Algorithm

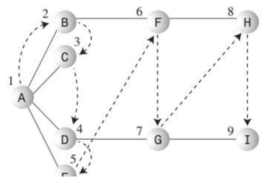
```
void DFS(Node startNode)
{
    for v in Nodes do
        v.visited = false;
    Stack s = new Stack;
    s.push(startNode);
    while (!s.empty())
    {
        x = s.pop();
        x.visited = true;
        /* process x */
        for y in x.children() do
            if (!y.visited)
            s.push(y);
    }
}
```

DFS Java Code

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

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

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

   }  // end dfs
```

Breadth First Search

- In the breadth-first search, on the other hand, the algorithm likes to stay as close as possible to the starting point.
- It visits all the vertices adjacent to the starting vertex, and only then goes further afield. This kind of search is implemented using a queue instead of a stack.



BFS

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.

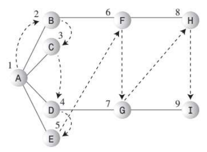
RULE 3

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

Breadth First Search

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

Breadth First Search



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

Comparison of BFS and DFS Algorithms

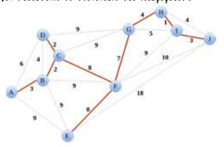
```
void DFS(Node startNode) {
   for v in Nodes do
         v.visited = false;
   Stack s = new Stack;

   s.push(startNode);
   while (!s.empty()) {
       x = s.pop();
       x.visited = true;
       /* process x */
       for y in x.children() do
         if (!y.visited) s.push(y);
}
```

```
void BFS(Node startNode) {
   for v in Nodes do
         v.visited = false;
   Queue s = new Queue;

   s.enqueue(startNode);
   while (!s.empty()) {
       x = s.dequeue();
       x.visited = true;
       /* process x */
       for y in x.children() do
         if (!y.visited) s.enqueue(y);
}
```

SCO 204 28

Minimum Spanning Tree

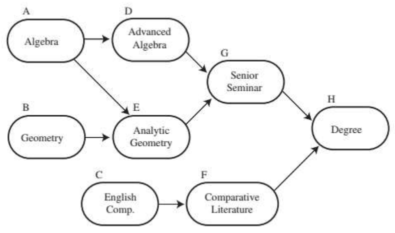
- In a weighted graph, you can sum the weights for all edges in a spanning tree and attempt to find a spanning tree that minimizes this sum
- There are several algorithms for finding a minimum spanning tree for a component.
- For example, to determine how an airline can service all cities, while minimizing the total length of the routes it needs to support



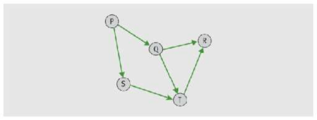
Topological Sort

- A directed acyclic graph (DAG) has an order among the vertices
- A topological order assigns a rank to each vertex such that the edges go from lower- to higher- ranked vertices
- **Topological sort**: process of finding and returning a topological order of vertices in a graph

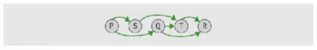
Example: Course Pre-requisites



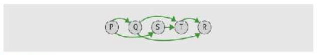
Topological Sort Example



[FIGURE 20.14] A graph of courses



[FIGURE 20.15] The first topological ordering of the graph



[FIGURE 20.16] The second topological ordering of the graph

Developing Graph ADT

- The implementation of the graph ADT shown here consists of the classes:
  - Graph
  - Vertex

Vertex Class

```
class Vertex
   {
   public char label;        // label (e.g. 'A')
   public boolean wasVisited;

   public Vertex(char lab)   // constructor
      {
      label = lab;
      wasVisited = false;
      }

   }  // end class Vertex
```

Graph Class

```
class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // array of vertices
   private int adjMat[][];      // adjacency matrix
   private int nVerts;          // current number of vertices

// ------------------------------------------------------------
-
   public Graph()               // constructor
      {
      vertexList = new Vertex[MAX_VERTS];
                                          // adjacency matrix
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // set adjacency
         for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
            adjMat[j][k] = 0;
      }  // end constructor

// ------------------------------------------------------------
```

Graph Class

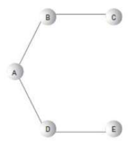
```
// ------------------------------------------------------------
-
   public void addVertex(char lab)    // argument is label
      {
      vertexList[nVerts++] = new Vertex(lab);
      }

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

// ------------------------------------------------------------
-
```

Graph Class

```
// ------------------------------------------------------------
-
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);

      }

// ------------------------------------------------------------
-

   }  // end class Graph
```

Example: Graph Creation

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

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

System.out.print("Visits: ");
theGraph.dfs();             // depth-first search
System.out.println();
```

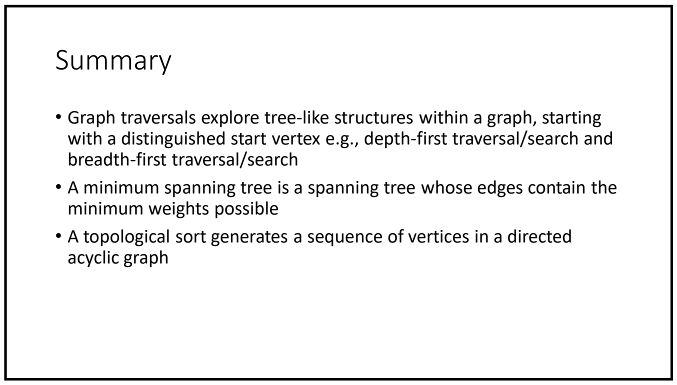


Summary

- Graphs have many applications
- A graph consists of one or more vertices (items) connected by one or more edges
- Path: Sequence of edges that allows one vertex to be reached from another vertex in the graph
- A graph is connected if there is a path from each vertex to every other vertex
- A graph is complete if there is an edge from each vertex to every other vertex

Summary

- A subgraph consists of a subset of a graph's vertices and a subset of its edges
- Connected component: a subgraph consisting of the set of vertices reachable from a given vertex
- Directed graphs allow travel along an edge in just one direction
- Edges can be labeled with weights, which indicate the cost of traveling along them
- Graphs have two common implementations:
- Adjacency matrix and adjacency list



Summary

- Graph traversals explore tree-like structures within a graph, starting with a distinguished start vertex e.g., depth-first traversal/search and breadth-first traversal/search
- A minimum spanning tree is a spanning tree whose edges contain the minimum weights possible
- A topological sort generates a sequence of vertices in a directed acyclic graph