Graphs

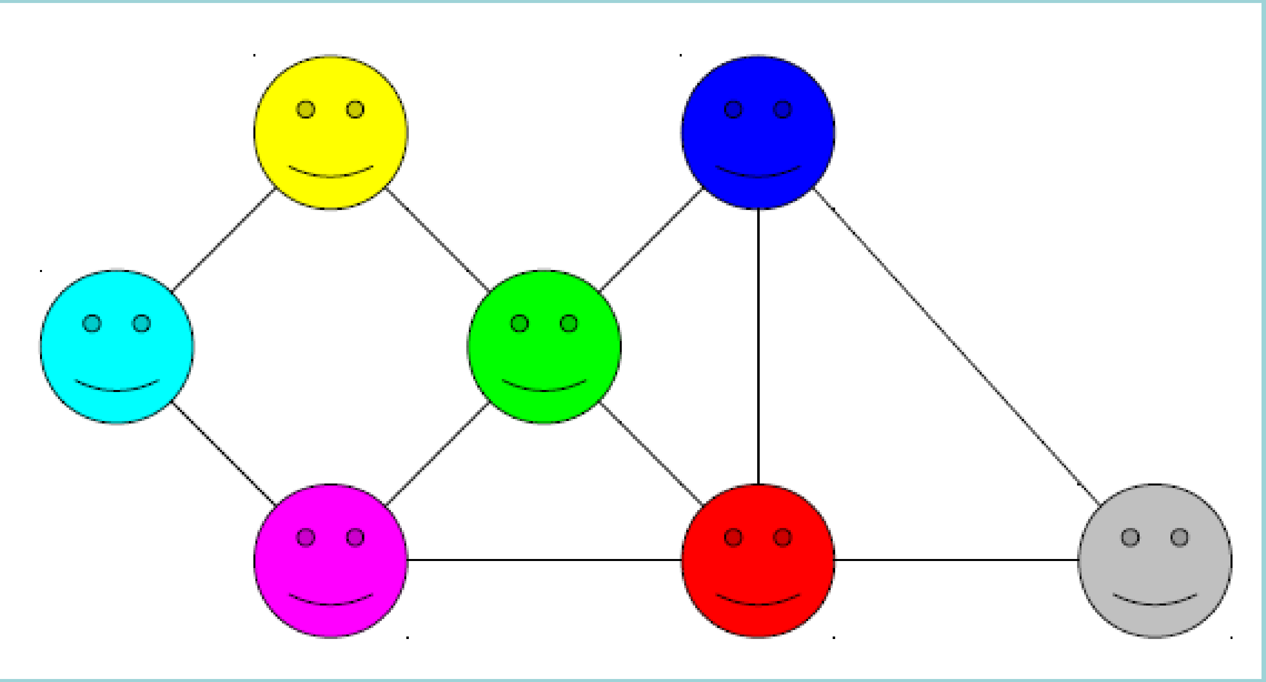
Objectives

- Learn about graphs
- Become familiar with the basic terminology of graph theory
- Discover how to represent graphs in computer memory
- Examine and implement various graph traversal algorithms
- Learn how to implement a shortest path algorithm
- Examine and implement the minimum spanning tree algorithm
- Explore topological sort
- Learn how to find Euler circuits in a graph

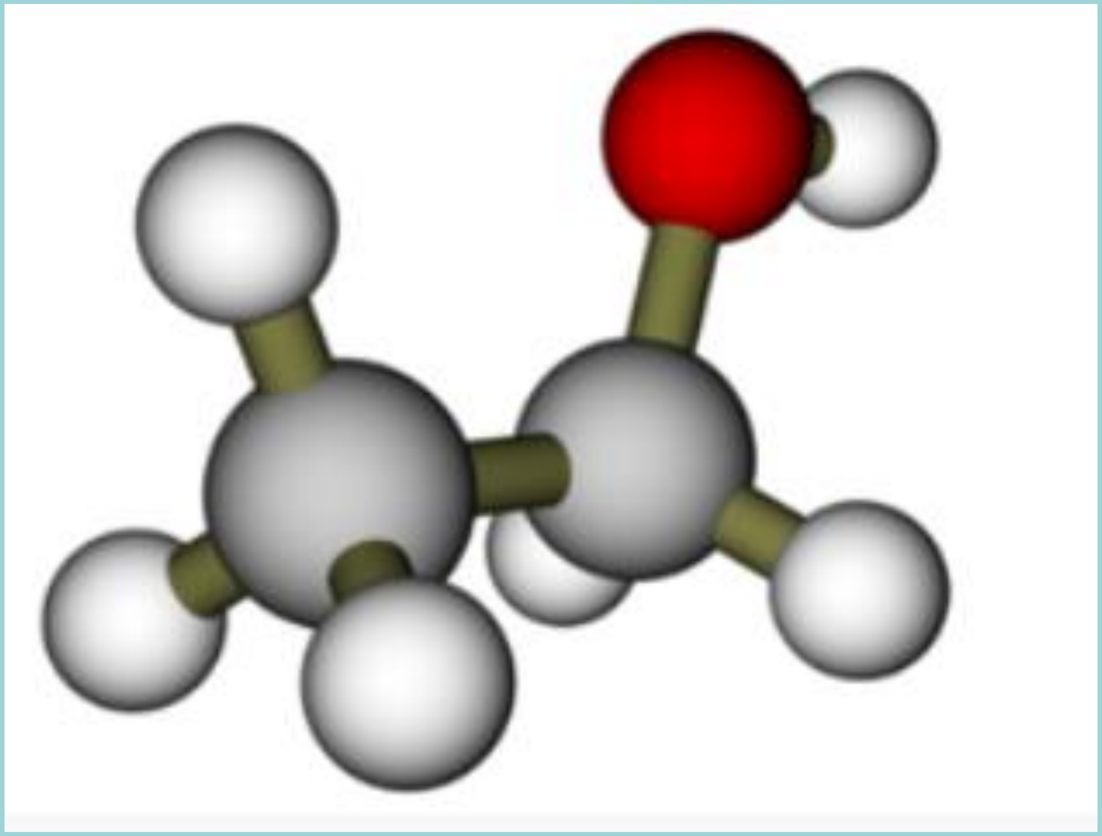
Introduction and Motivation

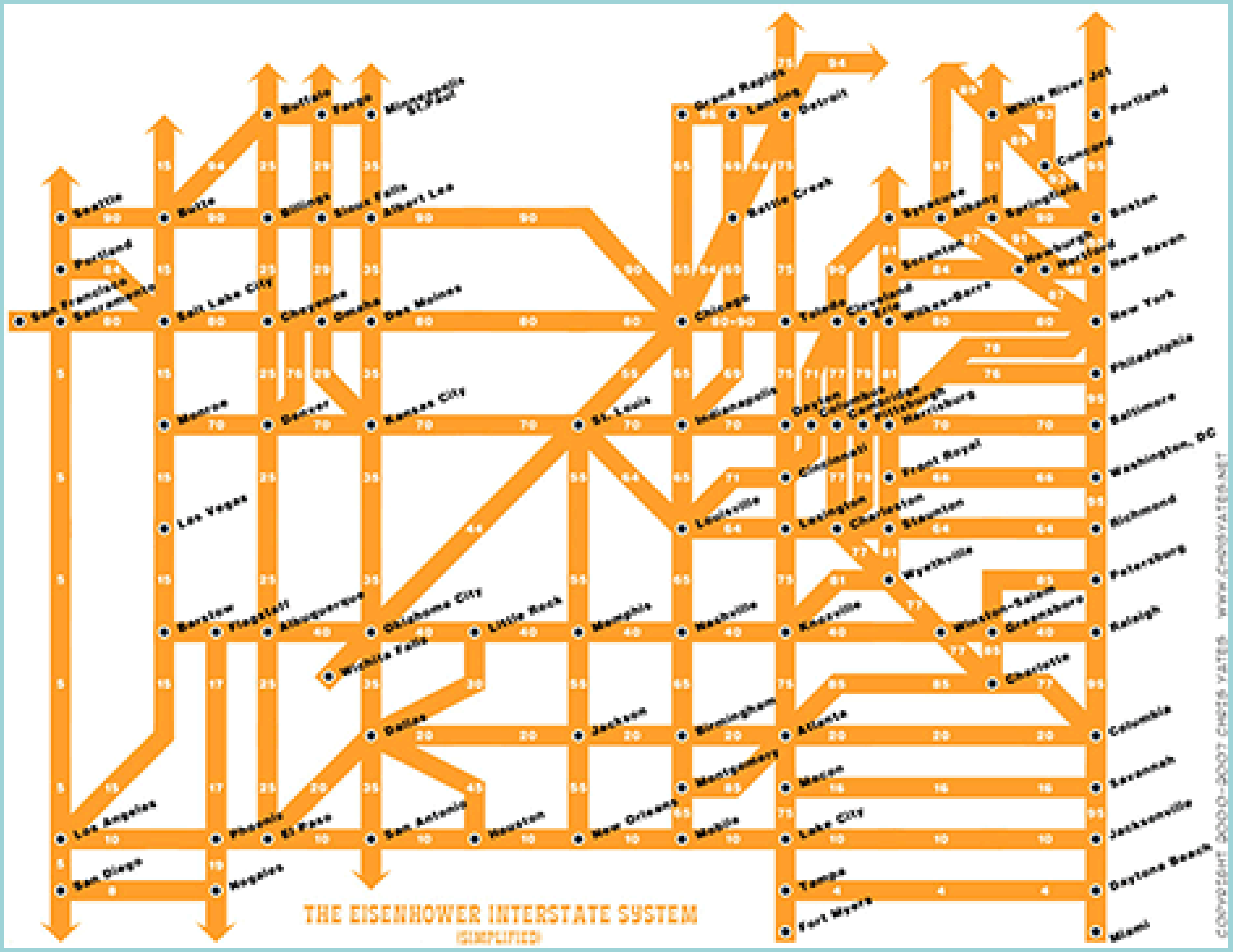
- Although trees are quite flexible, they have an inherent limitation in that they can only express hierarchical structures
- Fortunately, we can generalize a tree to form a ***graph***, in which this limitation is removed
- Informally, a graph is a collection of nodes and the connections between them
- Figure in next slide illustrates some examples of graphs; notice there is typically no limitation on the number of vertices or edges
- Consequently, graphs are extremely versatile and applicable to a wide variety of situations
- Graph theory has developed into a sophisticated field of study since its origins in the early 1700s

A Social Network



Chemical Bonds




THE EISENHOWER INTERSTATE SYSTEM
(SIMPLIFIED)
COPYRIGHT 2000-2007 CHRIS YATES
Seattle
Portland
San Francisco
Sacramento
Butte
Salt Lake City
Buffalo
Billings
Cheyenne
Fargo
Sioux Falls
Omaha
Minneapolis
St. Paul
Albert Lea
Des Moines
Monroe
Denver
Kansas City
Las Vegas
Barstow
Flagstaff
Albuquerque
Oklahoma City
Wichita Falls
Little Rock
Memphis
St. Louis
Chicago
Grand Rapids
Lansing
Detroit
Battle Creek
Indianapolis
Louisville
Nashville
Dallas
Jackson
Birmingham
Montgomery
Atlanta
Los Angeles
San Diego
Phoenix
Nogales
El Paso
San Antonio
Houston
New Orleans
Mobile
Macon
Lake City
Tampa
Fort Myers
Toledo
Cleveland
Erie
Dayton
Columbus
Cambridge
Pittsburgh
Harrisburg
Cincinnati
Lexington
Charleston
Staunton
Front Royal
Wytheville
Knoxville
Winston-Salem
Greensboro
Charlotte
Syracuse
Albany
Springfield
Scranton
Wilkes-Barre
Hartford
Concord
White River Jct
Portland
Boston
New Haven
New York
Philadelphia
Baltimore
Washington, DC
Richmond
Petersburg
Raleigh
Columbia
Savannah
Jacksonville
Daytona Beach
Miami

Graph Definitions and Notations

Graph Definitions and Notations

- Borrow definitions, terminology from set theory
- Subset
  - Set *Y* is a subset of *X*: $Y \subseteq X$
    - If every element of *Y* is also an element of *X*
- Intersection of sets *A* and *B*: $A \cap B$
  - Set of all elements that are in *A* and *B*
- Union of sets *A* and *B*: $A \cup B$
  - Set of all elements in *A* or in *B*
- Cartesian product: $A \times B$
  - Set of all ordered pairs of elements of *A* and *B*

Graph Definitions and Notations (cont'd.)

- Graph *G* pair $G = (V, E)$,
  - Where *V* is a finite nonempty set, called the set of vertices of *G*, and $E \subseteq V \times V$
    - Elements of *E* are pairs of elements of *V*.
  - *E* is called set of edges of *G*
    - *G* called trivial if it has only one vertex
- Graph *H* called subgraph of *G*
  - If $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$
  - Every vertex of *H*: vertex of *G*
  - Every edge in *H*: edge in *G*

Graph Definitions and Notations (cont'd.)

- A graph can be shown pictorially. The vertices are drawn as circles, and a label inside the circle represents the vertex.
- In an undirected graph, the edges are drawn using lines. Elements in set of edges of graph *G*: *un*ordered
- In a directed graph (digraph), the edges are drawn using arrows. Elements in set of edges of graph *G*: ordered

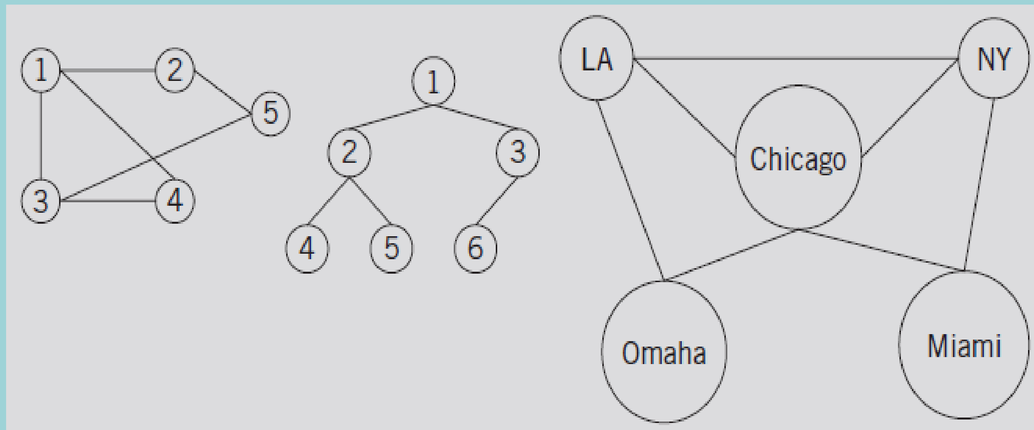


FIGURE 12-3 Various undirected graphs

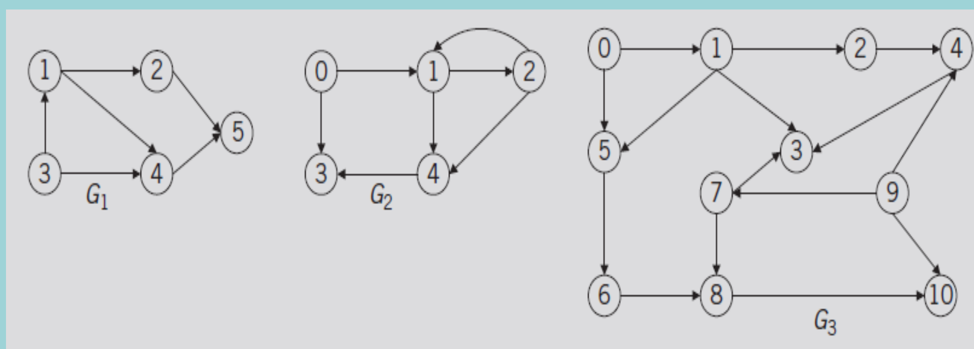


FIGURE 12-4 Various directed graphs

$V(G_1) = \{1, 2, 3, 4, 5\}$

$V(G_2) = \{0, 1, 2, 3, 4\}$

$V(G_3) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

$E(G_1) = \{(1, 2), (1, 4), (2, 5), (3, 1), (3, 4), (4, 5)\}$

$E(G_2) = \{(0, 1), (0, 3), (1, 2), (1, 4), (2, 1), (2, 4), (4, 3)\}$

$E(G_3) = \{(0, 1), (0, 5), (1, 2), (1, 3), (1, 5), (2, 4), (4, 3), (5, 6), (6, 8), (7, 3), (7, 8), (8, 10), (9, 4), (9, 7), (9, 10)\}$

Graph Definitions and Notations (cont'd.)

- Let u and v be two vertices in G
  - u and v adjacent
    - If edge from one to the other exists: $(u, v) \in E$
- Loop
  - Edge incident on a single vertex
- e_1 and e_2 called parallel edges
  - If two edges e_1 and e_2 associate with same pair of vertices $\{u, v\}$
- Simple graph
  - No loops, no parallel edges

Graph Definitions and Notations (cont'd.)

- Let $e = (u, v)$ be an edge in G
  - Edge e is incident on the vertices u and v
  - Degree of u written deg(u) or d(u)
    - Number of edges incident with u
- Each loop on vertex u
  - Contributes two to the degree of u
- u is called an even (odd) degree vertex
  - If the degree of u is even (odd)

Graph Definitions and Notations (cont'd.)

- Path from u to v
  - If sequence of vertices $u_1, u_2, \ldots, u_n$ exists
    - Such that $u = u_1$, $u_\text{n} = v$ and $(u_i, u_i + 1)$ is an edge for all $i = 1, 2, \ldots, n - 1$
- Vertices u and v called connected
  - If path from u to v exists
- Simple path
  - All vertices distinct (except possibly first, last)
- Cycle in G
  - Simple path in which first and last vertices are the same

Graph Definitions and Notations (cont'd.)

- G is connected
  - If path from any vertex to any other vertex exists
- Component of G
  - Maximal subset of connected vertices
- Let G be a directed graph and let u and v be two vertices in G
  - If edge from u to v exists: $(u, v) \in E$
    - u is adjacent to v
    - v is adjacent from u

Graph Definitions and Notations (cont’d.)

- Definitions of paths and cycles in *G*
  - Similar to those for undirected graphs
- *G* is strongly connected
  - If any two vertices in *G* are connected

Graph Representation

Graph Representation

- Graphs represented in computer memory
  - Two common ways
    - Adjacency matrices
    - Adjacency lists

Adjacency Matrices

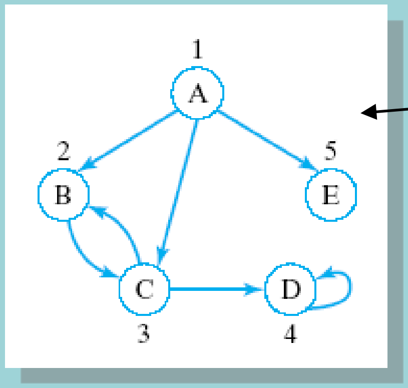
- Let G be a graph with n vertices where n > zero
- Let $V(G) = \{v_1, v_2, \ldots, v_n\}$
  - Adjacency matrix

$$A_G(i,j) = \begin{cases} 1 & \text{if } (v_i, v_j) \in E(G) \\ 0 & \text{otherwise} \end{cases}$$

$$A_{G_1} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad A_{G_2} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

Example: Adjacency matrix

Matrix [5][5] represents A, B, C, D, E vertices



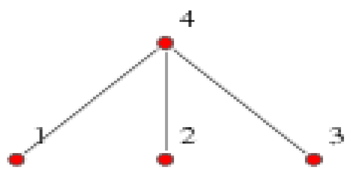
columns ***j***

rows ***i***

	1	*2*	*3*	*4*	*5*
1	0	1	1	0	1
2	0	0	1	0	0
3	0	0	0	1	0
4	0	0	1	0	0
5	0	0	0	0	0

- Entry [1, 5] set to true
- Edge from vertex 1 to vertex 5

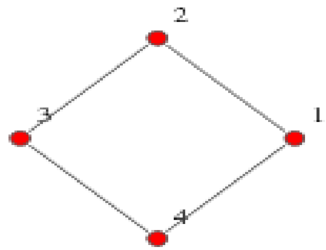
More Examples of Adjacency Matrix



$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Claw graph

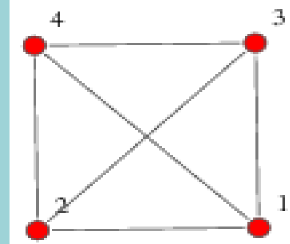
Matrix[4][4]



$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Cycle graph

Matrix[4][4]



$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

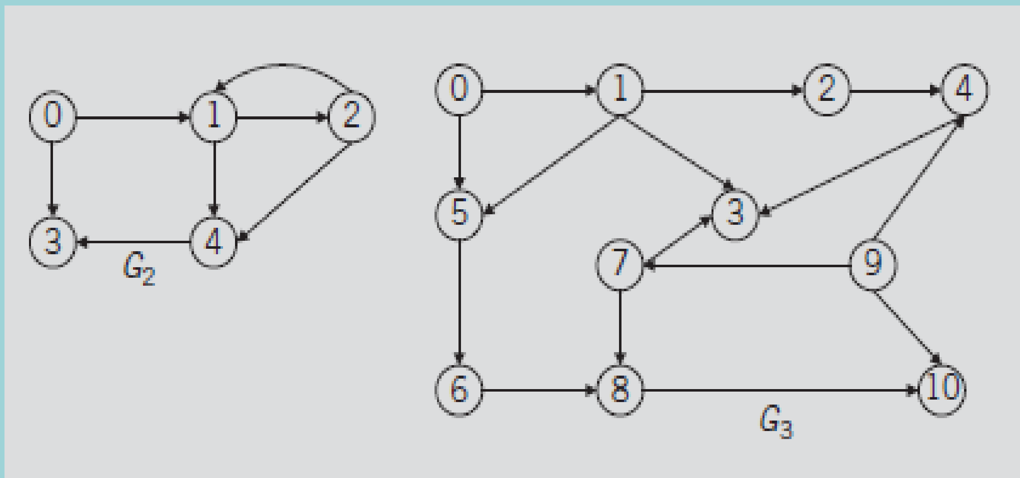
Complete graph

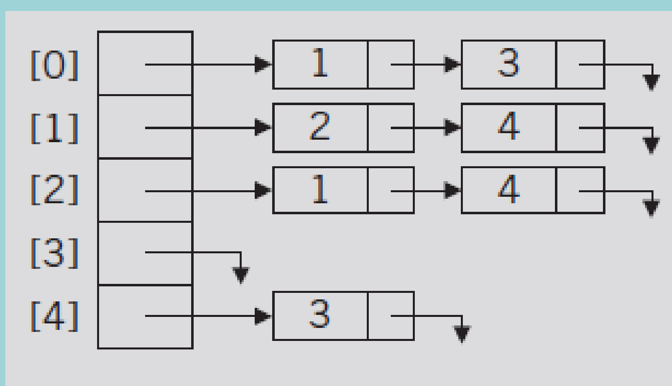
Matrix[4][4]

Adjacency Lists

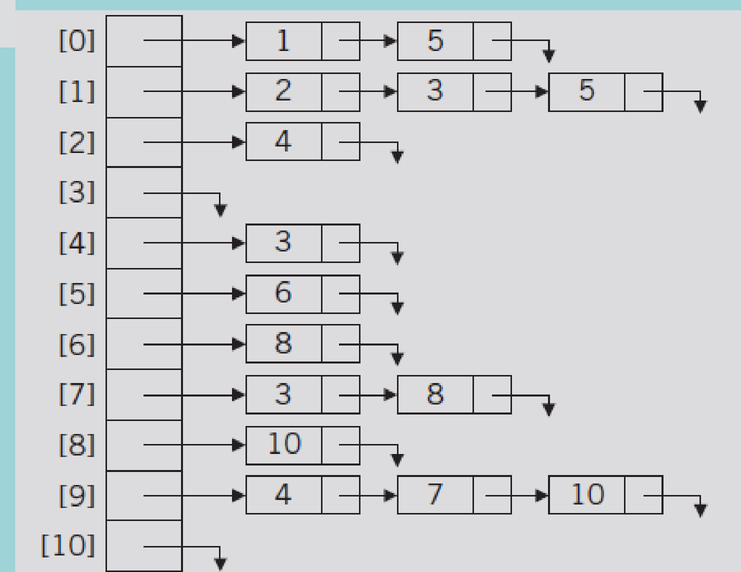
- Given:
  - Graph G with n vertices, where $n > zero$
  - $V(G) = \{v_1, v_2, ..., v_n\}$
- For each vertex v: linked list exists
  - Linked list node contains vertex u: $(v, u) \in E(G)$
- Use array A, of size n, such that $A[i]$
  - Reference variable pointing to first linked list node containing vertices to which v_i adjacent
- Each node has two components: vertex, link
  - Component vertex
    - Contains index of vertex adjacent to vertex i

Adjacency Lists- Examples





Adjacency list of graph *G2*



Adjacency list of graph *G3*

Operations on Graph

Operations on Graphs

- Commonly performed operations
  - Create graph
    - Store graph in computer memory using a particular graph representation
  - Clear graph
    - Makes graph empty
  - Determine if graph is empty
  - Traverse graph
  - Print graph

Operations on Graphs (cont'd.)

- Graph representation in computer memory
  - Depends on specific application
- Use linked list representation of graphs
  - For each vertex *v*
    - Vertices adjacent to *v* (directed graph: called immediate successors)
    - Stored in the linked list associated with *v*
- Managing data in a linked list
  - Use `class unorderedLinkedList`
- Labeling graph vertices
  - Depends on specific application

Graphs as ADTs

Defines a graph as an ADT

– Class specifying basic operations to implement a graph

- Definitions of the functions of the `class graphType`

```
bool graphType::isEmpty() const
{
    return (gSize == 0);
}
```

Graphs as ADTs (cont'd.)

- Function `createGraph`
  - Implementation
    - Depends on how data input into the program
- Function `clearGraph`
  - Empties the graph
    - Deallocates storage occupied by each linked list
    - Sets number of vertices to zero

Graph Traversals

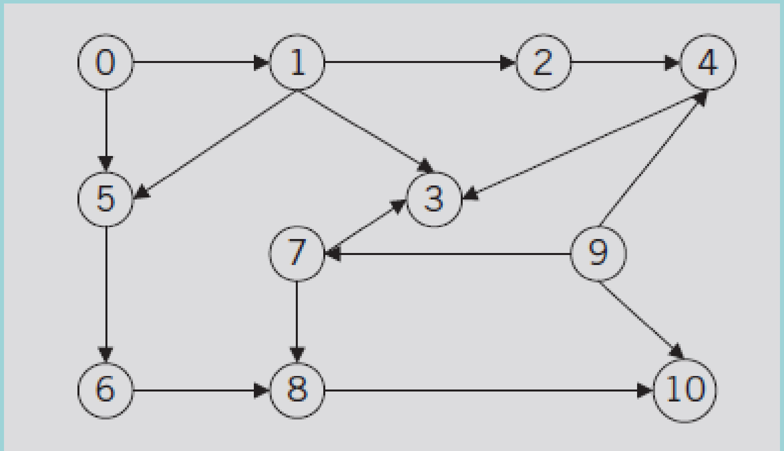
- Processing a graph
  - Requires ability to traverse the graph
- Traversing a graph
  - Similar to traversing a binary tree
    - A bit more complicated
- Two most common graph traversal algorithms
  - Depth first traversal
  - Breadth first traversal

Depth-First Traversal

Depth First Traversal

- Similar to binary tree preorder traversal – Goes as far as possible from a vertex before backing up.
- General algorithm

```
for each vertex, v, in the graph
        if v is not visited
                start the depth first traversal at v
```



Directed graph

Depth First Traversal (cont'd.)

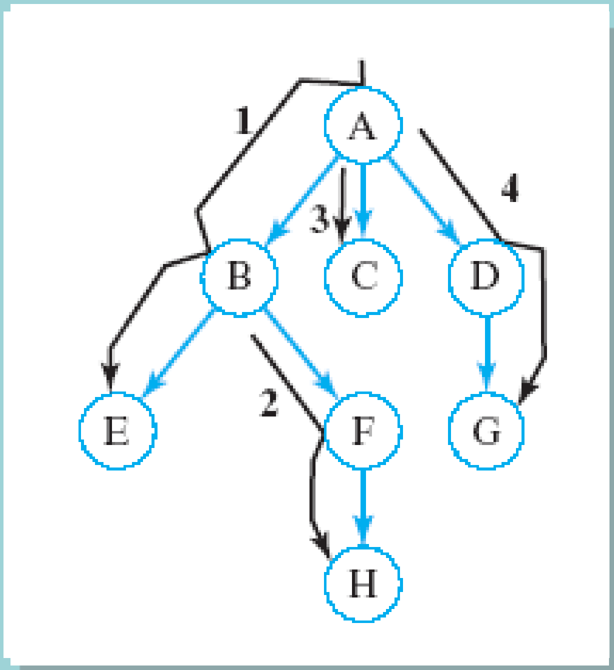
- General algorithm for depth first traversal at a given node *v*
  - Recursive algorithm

```
1. mark node v as visited
2. visit the node
3. for each vertex u adjacent to v
        if u is not visited
           start the depth first traversal at u
```

Depth-First Search

- Start from node 1
- What is a sequence of nodes which would be visited in DFS?

A, B, E, F, H, C, D, G



Depth First Traversal (cont'd.)

- Function `dft` implements algorithm

```
void graphType::dft(int v, bool visited[])
{
    visited[v] = true;
    cout << " " << v << " ";  //visit the vertex

    // declare graphIt to be an iterator to be used
    // to traverse a linked list to which graph[v] points
    linkedListIterator<int> graphIt;

        //for each vertex adjacent to v
    for (graphIt = graph[v].begin(); graphIt != graph[v].end();
                                     ++graphIt)
    {
        // *graphIT returns the label of the vertex
        int w = *graphIt;
        if (!visited[w])
            dft(w, visited);
    } //end while
} //end dft
```

Depth First Traversal (cont'd.)

- Function `depthFirstTraversal`
  - Implements depth first traversal of the graph

```
void graphType::depthFirstTraversal()
{
    bool *visited; //pointer to create the array to keep
                   //track of the visited vertices
    visited = new bool[gSize];

    for (int index = 0; index < gSize; index++)
        visited[index] = false;

        //For each vertex that is not visited, do a depth
        //first traverssal
    for (int index = 0; index < gSize; index++)
        if (!visited[index])
            dft(index,visited);
    delete [] visited;
} //end depthFirstTraversal
```

Depth First Traversal (cont'd.)

- Function `depthFirstTraversal`
  - Performs a depth first traversal of entire graph
- Function `dftAtVertex`
  - Performs a depth first traversal at a given vertex

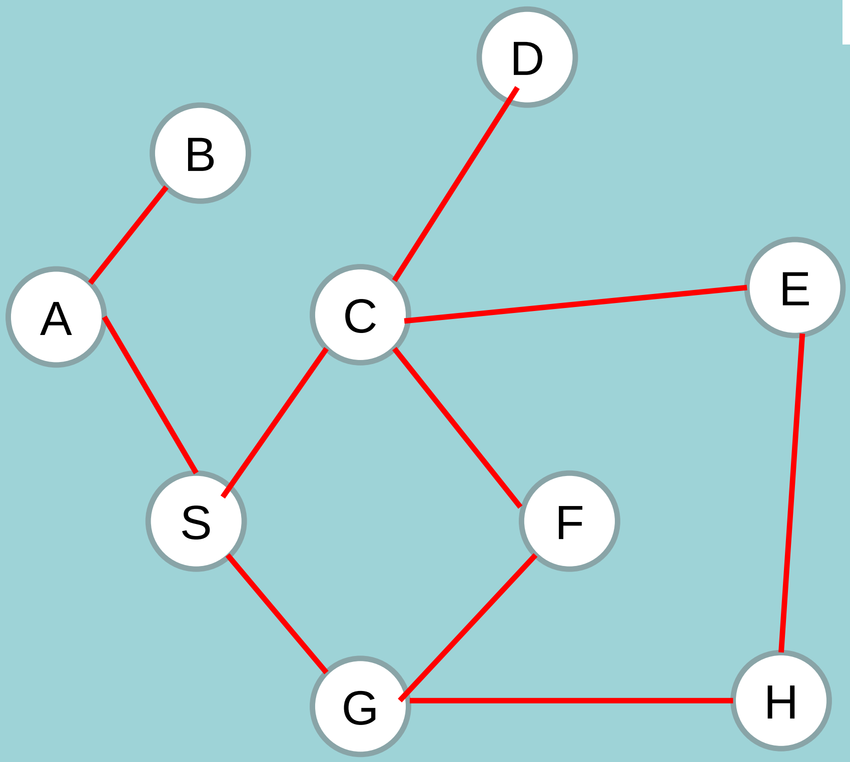
```
void graphType::dftAtVertex(int vertex)
{
    bool *visited;

    visited = new bool[gSize];
    for (int index = 0; index < gSize; index++)
        visited[index] = false;

    dft(vertex, visited);

    delete [] visited;
} // end dftAtVertex
```

Breadth-First Traversal

Breadth First Traversal

- Similar to traversing binary tree level-by-level
  - Visits all vertices adjacent to a vertex before going forward. BFT visits nodes by level.
    - Start from a given vertex of v; visit all neighbors of first neighbor of v
    - Then visit all neighbors of second neighbor x of v …

- Start from node A
- What is a sequence of nodes which would be visited in DFS?



A, B, S, C, D, E, F, G, H

Breadth First Traversal

- General search algorithm
  - Breadth first search algorithm with a queue

```
for each vertex v in the graph
        if v is not visited
                add v to the queue // start bf search at v
Mark v as visited
While the queue is not empty
        remove vertex u from the queue
        retrieve the vertices adjacent to u
        for each vertex w that is adjacent to u
                add w to the queue
                mark w as visited
```

- C++ function implements breadFirstTraversal this algorithm

```cpp
void graphType::breadthFirstTraversal()
{
    linkedQueueType<int> queue;

    bool *visited;
    visited = new bool[gSize];

    for (int ind = 0; ind < gSize; ind++)
        visited[ind] = false;   //initialize the array
                                //visited to false
    linkedListIterator<int> graphIt;
    for (int index = 0; index < gSize; index++)
        if (!visited[index])
        {
            queue.addQueue(index);
            visited[index] = true;
            cout << " " << index << " ";

            while (!queue.isEmptyQueue())
            {
                int u = queue.front();
                queue.deleteQueue();

                for (graphIt = graph[u].begin();
                     graphIt != graph[u].end(); ++graphIt)
                {
                    int w = *graphIt;
                    if (!visited[w])
                    {
                        queue.addQueue(w);
                        visited[w] = true;
                        cout << " " << w << " ";
                    }
                }
            } //end while
        }

    delete [] visited;
} //end breadthFirstTraversal
```

Shortest Path Algorithm

Shortest Path Algorithm

- Weight of the graph
  - Nonnegative real number assigned to the edges connecting to vertices.
- Weighted graphs
  - When a graph uses the weight to represent the distance between two places
- Weight of the path P
  - Given G as a weighted graph with vertices u and v in G and P as a path in G from u to v
    - Sum of the weights of all the edges on the path
- Shortest path: path with the smallest weight

Shortest Path Algorithm (cont'd.)

- Shortest path algorithm space (greedy algorithm)
- Using inheritance, we extend the definition of the `class graphType` by:
  - Extend definition of `class graphType`
  - Adds function `createWeightedGraph` to create graph and weight matrix associated with the graph
  - Call `class weightedGraphType`

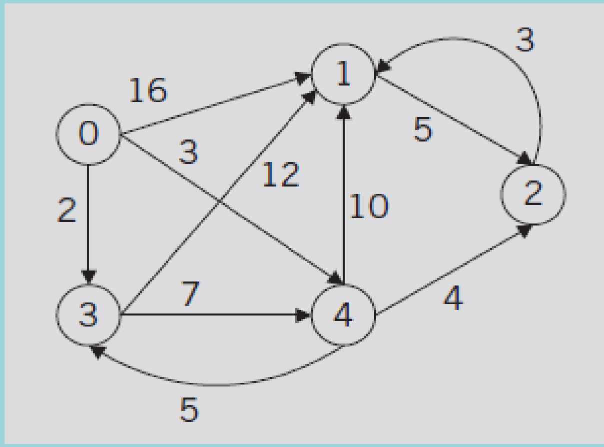
Sample of class weightedGraphType

```
class weightedGraphType: public graphType
{
public:
    void createWeightedGraph();
      //Function to create the graph and the weight matrix.
      //Postcondition: The graph using adjacency lists and
      //    its weight matrix is created.

    void shortestPath(int vertex);
      //Function to determine the weight of a shortest path
      //from vertex, that is, source, to every other vertex
      //in the graph.
      //Postcondition: The weight of the shortest path from vertex
      //    to every other vertex in the graph is determined.

    void printShortestDistance(int vertex);
      //Function to print the shortest weight from the vertex
      //specified by the parameter vertex to every other vertex in
      //the graph.
      //Postcondition: The weight of the shortest path from vertex
      //    to every other vertex in the graph is printed.

    weightedGraphType(int size = 0);
      //Constructor
      //Postcondition: gSize = 0; maxSize = size;
      // graph is an array of pointers to linked lists.
      // weights is a two-dimensional array to store the weights of the edges.
      //    smallestWeight is an array to store the smallest weight
      //    from source to vertices.

    ~weightedGraphType();
      //Destructor
      //The storage occupied by the vertices and the arrays
      //weights and smallestWeight is deallocated.

protected:
    double **weights;   //pointer to create weight matrix
    double *smallestWeight; //pointer to create the array to store
                     //the smallest weight from source to vertices
};
```

Shortest Path

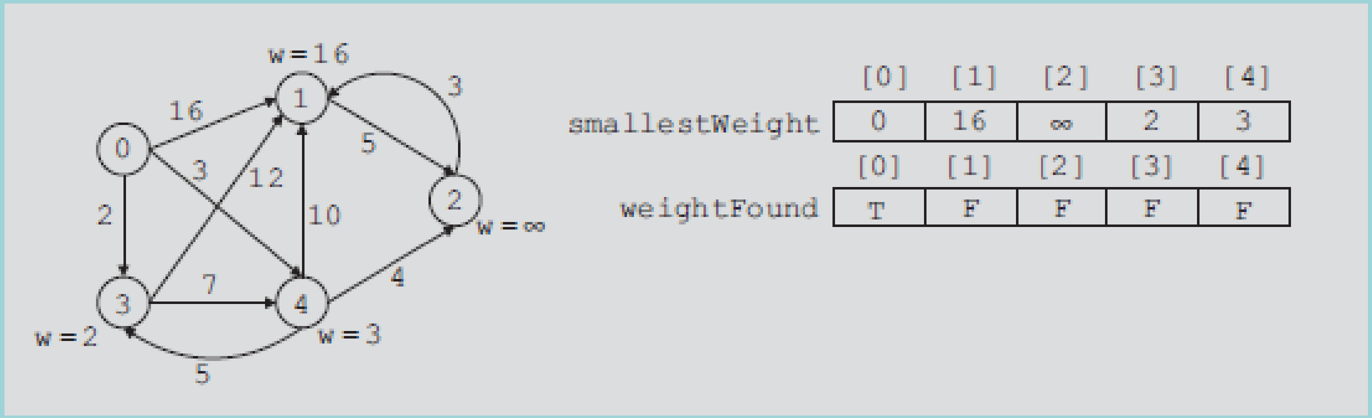
- General algorithm
  1. Initialize array *smallestWeight*
     - smallestWeight[*u*] = weights[*vertex*, *u*]
  2. Set smallestWeight[*vertex*] = zero
  3. Find vertex *v* closest to vertex where shortest path is not determined
  4. Mark *v* as the (next) vertex for which the smallest weight is found
  5. For each vertex *w* in *G*, such that the shortest path from vertex to w has not been determined and an edge (*v*, *w*) exists
     - If weight of the path to *w* via *v* smaller than its current weight
     - Update weight of *w* to the weight of *v* + weight of edge (*v*, *w*)

Shortest Path (cont'd.)

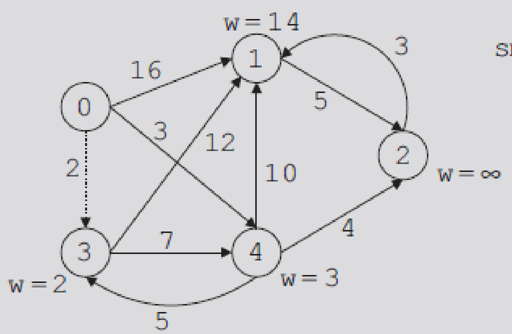


Weighted graph *G*

1. Initialize array *smallestWeight*
   - smallestWeight[*u*] = weights[*vertex*, *u*]
2. Set smallestWeight[*vertex*] = zero

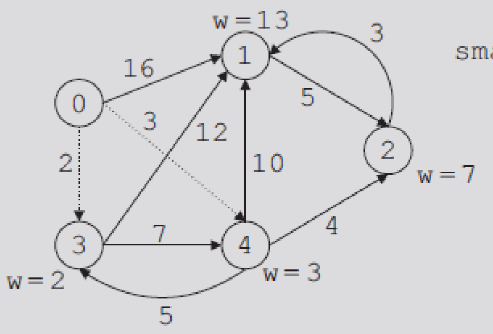


Shortest Path (cont'd.)



	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	14	∞	2	3
weightFound	T	F	F	T	F

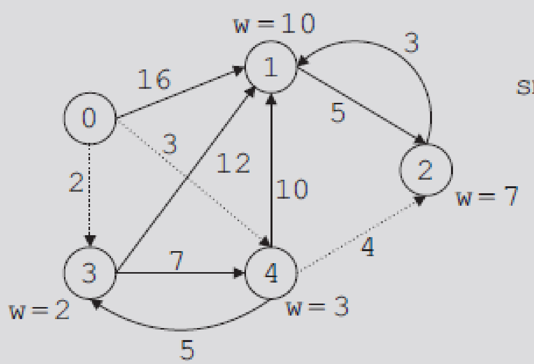
Graph after the first iteration of Steps 3 to 5



	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	13	7	2	3
weightFound	T	F	F	T	T

Graph after the second iteration of Steps 3 to 5

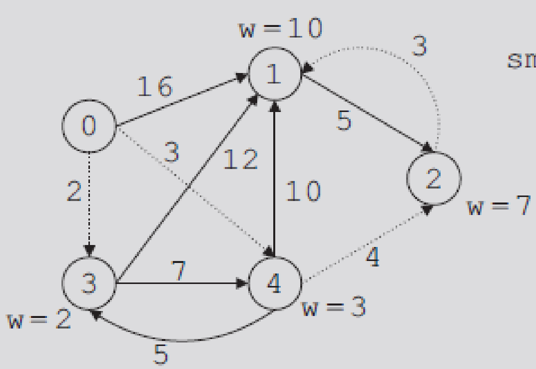
Shortest Path (cont'd.)



	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	10	7	2	3

	[0]	[1]	[2]	[3]	[4]
weightFound	T	F	T	T	T

Graph after the third iteration of Steps 3 to 5



	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	10	7	2	3

	[0]	[1]	[2]	[3]	[4]
weightFound	T	T	T	T	T

Graph after the fourth iteration of Steps 3 through 5

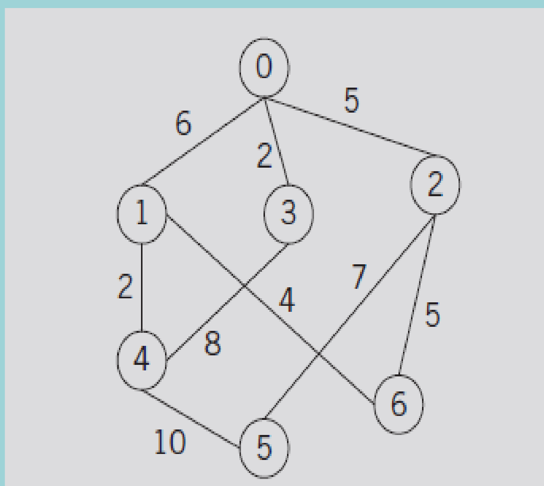
```cpp
void weightedGraphType::shortestPath(int vertex)
{
    for (int j = 0; j < gSize; j++)
        smallestWeight[j] = weights[vertex][j];

    bool *weightFound;
    weightFound = new bool[gSize];

    for (int j = 0; j < gSize; j++)
        weightFound[j] = false;

    weightFound[vertex] = true;
    smallestWeight[vertex] = 0;

    for (int i = 0; i < gSize - 1; i++)
    {
        double minWeight = DBL_MAX;
        int v;

        for (int j = 0; j < gSize; j++)
            if (!weightFound[j])
                if (smallestWeight[j] < minWeight)
                {
                    v = j;
                    minWeight = smallestWeight[v];
                }

        weightFound[v] = true;

        for (int j = 0; j < gSize; j++)
            if (!weightFound[j])
                if (minWeight + weights[v][j] < smallestWeight[j])
                    smallestWeight[j] = minWeight + weights[v][j];
    } //end for
} //end shortestPath
```

C++ function `shortestPath` implements previous algorithm

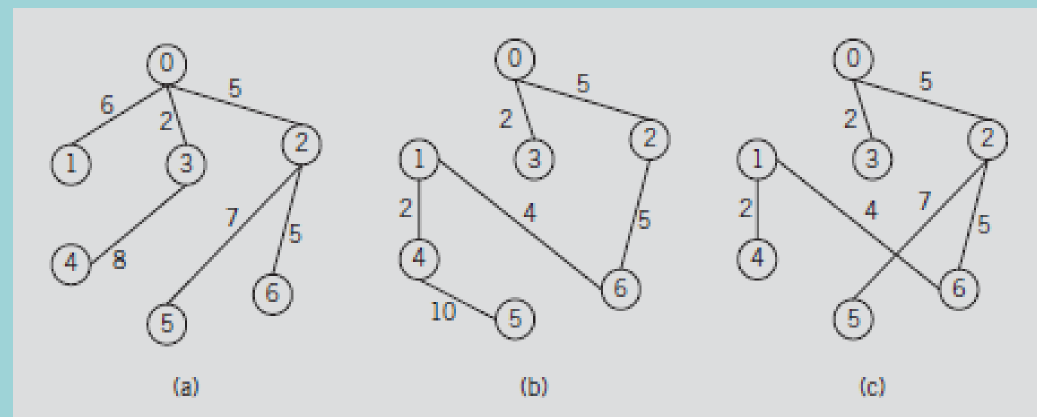
- Records only the weight of the shortest path from the source to a vertex

Minimum Spanning Tree

- Airline connections of a company
  - Between seven cities



Airline connections between cities and the cost factor of maintaining the connections



Possible solutions (a, b and c) if company must shutdown maximum number of connections.
Total cost of (a) = 33
Total cost of (b) = 28
Total cost of c) = 25
The desired solution would be (c) because it gives the lowest cost.
Graphs a, b and c are called spanning trees of the original graph.

Minimum Spanning Tree - Terminology

- Free tree *T*
  - Simple graph
  - If *u* and *v* are two vertices in *T*
    - Unique path from *u* to *v* exists
- Rooted tree
  - Tree with particular vertex designated as a root

Minimum Spanning Tree -Terminology

- Weighted tree T
  - Weight assigned to edges in T
  - Weight denoted by $W(T)$: sum of weights of all the edges in T
- Spanning tree T of graph G
  - T is a subgraph of G such that $V(T) = V(G)$

Minimum Spanning Tree -Terminology

- Spanning tree theorem
  - A graph *G* has a spanning tree if and only if *G* is connected
  - From this theorem, it follows that to determine a spanning tree of a graph
    - Graph must be connected
- Minimum (minimal) spanning tree of *G*
  - Spanning tree with the minimum weight

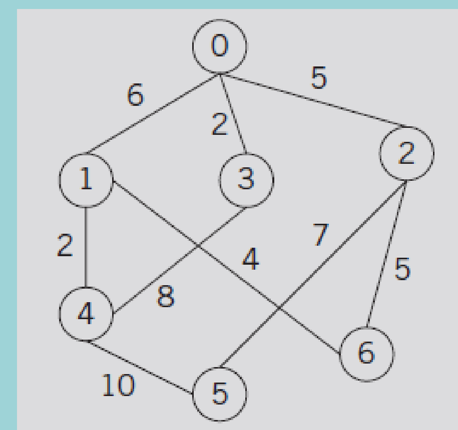
Minimum Spanning Tree (MST) - Algorithm

- Two well-known algorithms for finding a minimum spanning tree of a graph
  - Prim's algorithm
    - Builds the tree iteratively by adding edges until a minimum spanning tree obtained
  - Kruskal's algorithm – not cover in this lecture

MST – Prim's algorithm

- General form of Prim's algorithm

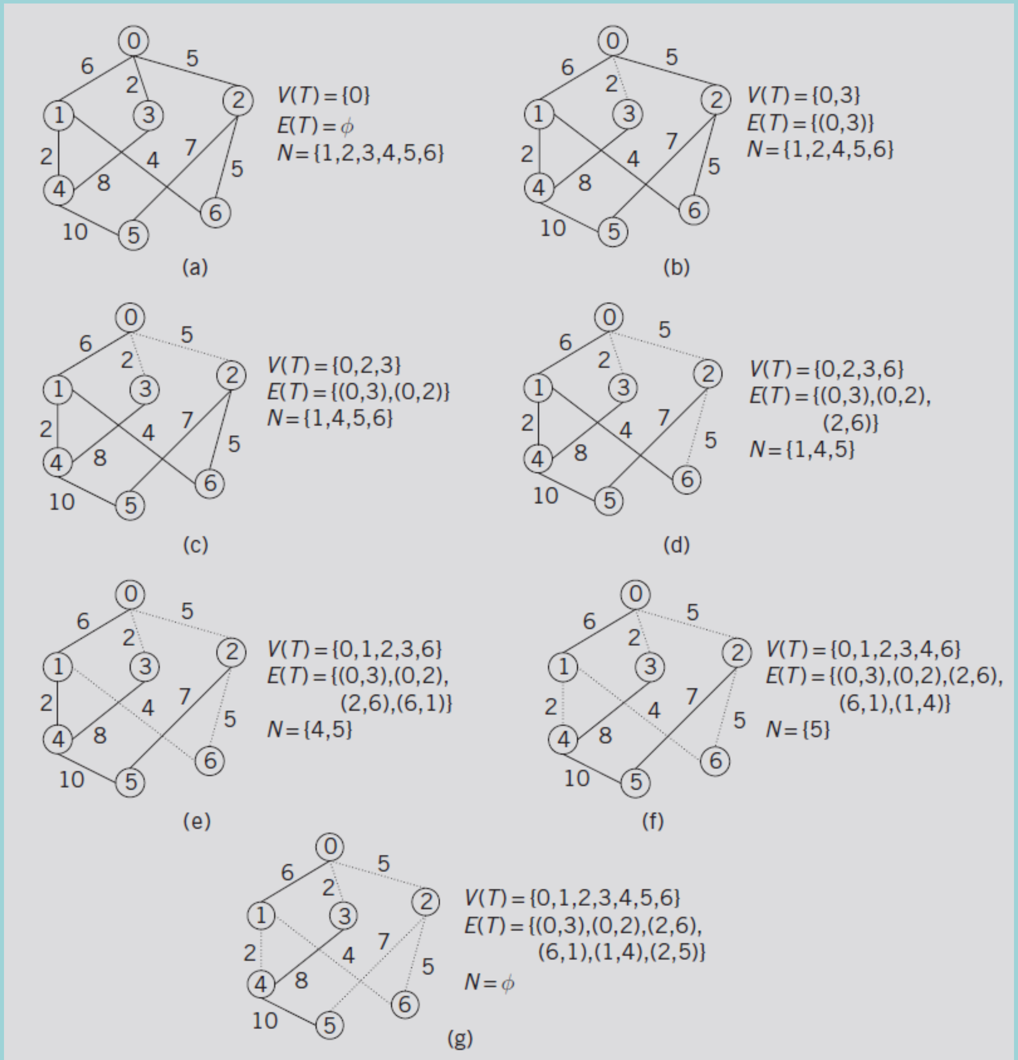
```
1.  Set V(T) = {source}
2.  Set E(T) = empty
3.  for i = 1 to n
    3.1.  minWeight = infinity;
    3.2.  for j = 1 to n
              if v_j is in V(T)
                for k = 1 to n
                 if v_k is not in T and weight[v_j, v_k] < minWeight
                 {
                     endVertex = v_k;
                     edge = (v_j, v_k);
                     minWeight = weight[v_j, v_k];
                 }
    3.3.  V(T) = V(T) ∪ {endVertex};
    3.4.  E(T) = E(T) ∪ {edge};
```



MST - Prim's algorithm Implementation

- `class msTreeType` defines spanning tree as an ADT
- C++ function `minimumSpanning` implementing Prim's algorithm
- Prim's algorithm given in this section: $O(n^3)$
  - Possible to design Prim's algorithm order $O(n^2)$

- See function `printTreeAndWeight` code
- See constructor and destructor code



Graph *G*, *V*(*T*), *E*(*T*), and *N* after Steps 1 and 2 execute

Topological Order

- Topological ordering of $V(G)$
  - Linear ordering $v_{i1}, v_{i2}, \ldots, v_{in}$ of the vertices such that
    - If v_{ij} is a predecessor of v_{ik}, $j \neq k$, $1 \leq j \leq n$, $1 \leq k \leq n$
    - Then v_{ij} precedes v_{ik}, that is, $j < k$ in this linear ordering
- Algorithm topological order
  - Outputs directed graph vertices in topological order
  - Assume graph has no cycles
    - There exists a vertex v in G such that v has no successor
    - There exists a vertex u in G such that u has no predecessor

Topological Order (cont'd.)

- Topological sort algorithm
  - Implemented with the depth first traversal or the breadth first traversal
- Extend `class graphType` definition (using inheritance)
  - Implement breadth first topological ordering algorithm
    - Called `class topologicalOrderType`
  - See code on pages 714-715
    - Illustrating class including functions to implement the topological ordering algorithm

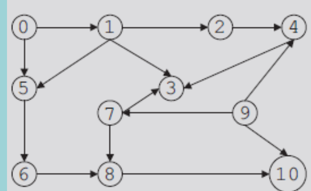
Breadth First Topological Ordering

- General algorithm

1. Create the array `predCount` and initialize it so that `predCount[i]` is the number of predecessors of the vertex v_i.
2. Initialize the queue, say `queue`, to all those vertices v_k so that `predCount[k]` is 0. (Clearly, `queue` is not empty because the graph has no cycles.)
3. `while` the queue is not empty
   - 3.1. Remove the front element, `u`, of the queue.
   - 3.2. Put `u` in the next available position, say `topologicalOrder[topIndex]`, and increment `topIndex`.
   - 3.3. For all the immediate successors `w` of `u`,
     - 3.3.1. Decrement the predecessor count of `w` by `1`.
     - 3.3.2. `if` the predecessor count of `w` is 0, add `w` to `queue`.

Breadth First Topological Ordering (cont'd.)

- `Breadth First Topological order`
  - 0 9 1 7 2 5 4 6 3 8 10



predCount

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
0	1	1	3	2	2	1	1	2	0	2

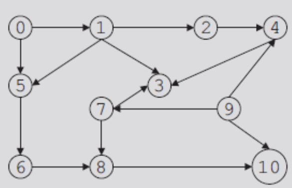
topologicalOrder

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

queue

0, 9

Arrays `predCount`, `topologicalOrder`, and `queue` after Steps 1 and 2 execute



predCount

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
0	0	1	3	2	1	1	1	2	0	2

topologicalOrder

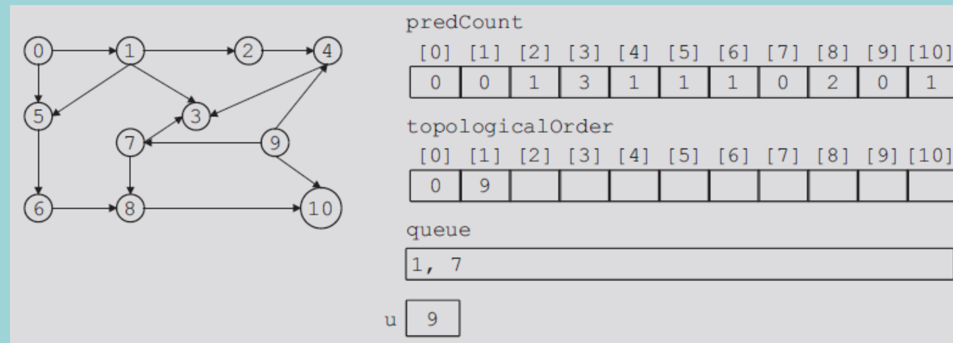
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
0										

queue

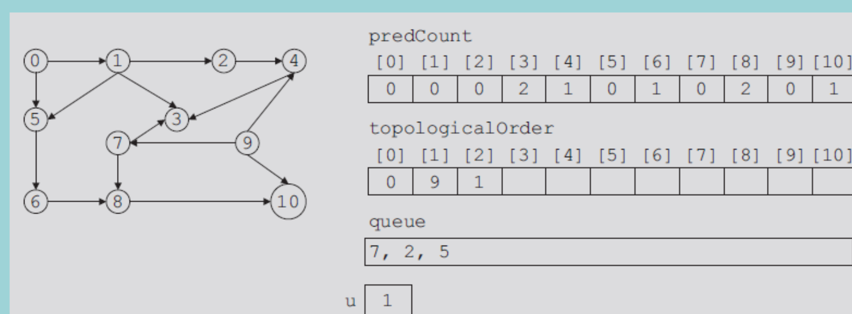
9, 1

u 0

Arrays `predCount`, `topologicalOrder`, and `queue` after the first iteration of Step 3



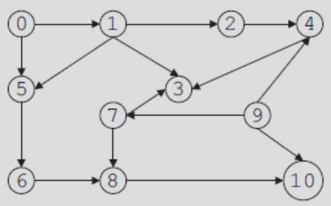
Arrays `predCount`, `topologicalOrder`, and `queue` after the second iteration of Step 3



Arrays `predCount`, `topologicalOrder`, and `queue` after the third iteration of Step 3

Breadth First Topological Ordering (cont'd.)

- See code on pages 718-719
  - Function implementing breadth first topological ordering algorithm



predCount

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
0	0	0	0	0	0	0	0	0	0	0

topologicalOrder

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
0	9	1	7	2	5	4	6	3	8	10

queue

Arrays `predCount`, `topologicalOrder`, and `queue` after Step 3 executes

Summary

- Many types of graphs
  - Directed, undirected, subgraph, weighted
- Graph theory borrows set theory notation
- Graph representation in memory
  - Adjacency matrices, adjacency lists
- Graph traversal
  - Depth first, breadth first
- Shortest path algorithm
- Prim's algorithm