CHAPTER 8

Graph and Their Applications

Introduction & Graph theory Terminology

- A graph G consists of two things:
  1. A set V of elements called nodes (or points or vertices).
  2. A set E of edges such that each edge e in E is identified with a unique(unordered) pair [u, v] of nodes in V, denoted by e=[u, v].
- Graph is indicated by: G=(V, E).
- Endpoints: Suppose e=[u, v] is an edge. The nodes u and v are called the endpoints of e.
- Adjacent nodes/neighbors: Suppose e=[u, v] is an edge. The nodes u and v are said to be adjacent nodes or neighbors.
- Degree of a node: The degree of a node u, written as deg(u), is the no. of edges containing u.
- Isolated node: If deg(u)=0, that is if u does not belong to any edge, then u is called an isolated node.

Introduction & Graph theory Terminology

- Path: A path P of length n from a node u to node v is defined as a sequence of n+1 nodes such that $u=v_0$; v_{i-1} is adjacent to v_i for i=1, 2, …,n; and $v_n=v$.

  $P=(v_0, v_1, v_2, \ldots, v_n)$.

- Closed path: The path P is said to be closed if $v_0 = v_n$.

- Simple path: The path P is said to be simple if all the nodes are distinct, with the exception that v_0 may equal v_n; that is, P is simple if the nodes $v_0, v_1, \ldots, v_{n-1}$ are distinct and the nodes $v_1, v_2, \ldots v_n$ are distinct.

- Cycle: A cycle is a closed simple path with length 3 or more.

- Connected Graph: A graph G is said to connected if there is a path between any two of its nodes.

Propositions

- A graph G is connected if and only if there is a simple path between any two nodes in G.
- Complete: A graph G is said to be complete if every node u in G is adjacent to every other node v in G. Clearly such graphs are connected. A complete graph with n nodes will have n(n-1)/2 edges.
- Tree graph or free tree: A connected graph T without any cycles is called a tree graph or free tree or simply a tree
- Labeled graph: A graph G is said to be labeled if its edges are assigned data.
- Weighted graph: A graph G is said to be weighted if each edge e in g is assigned a nonnegative value w(e) called the weight or length of e.

Propositions

- By default, any graph G is weighted graph and the weight of each edge is w(e)=1.

- Multigraph: A graph G is called a multigraph if the graph allow either multiple edges or loops:

  1. Multiple edges: Distinct edges e and e` are called multiple edges if they connect the same endpoints, that is, if e=[u, v] and e`=[u, v].

  2. Loops: An edges e is called a loop if it has identical endpoints, that is, if e=[u, u].

- Finite graph: A multigraph is said to be finite if it has a finite no. of nodes and a finite no. of edges.

Graph Examples

Example:

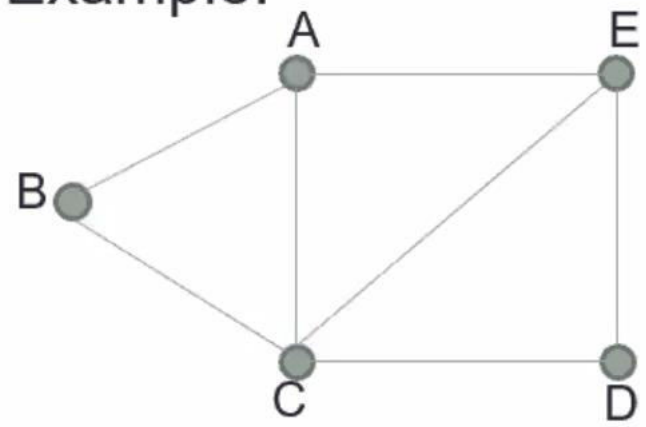


Fig a: Graph

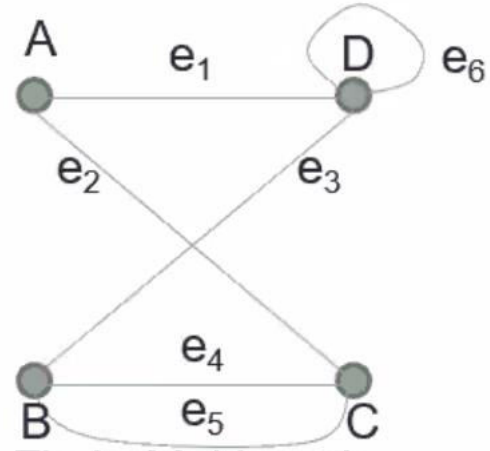


Fig b: Multigraph

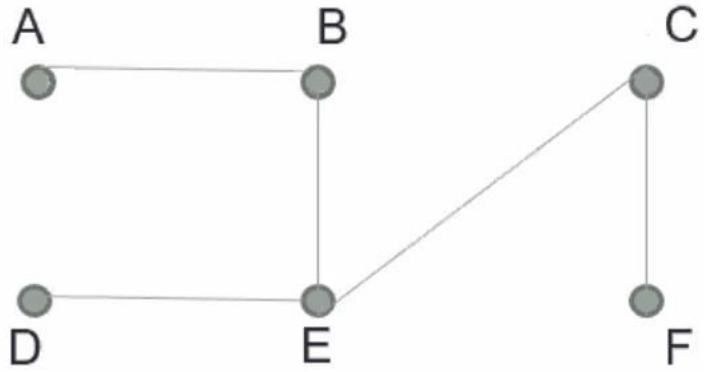


Fig c: Tree

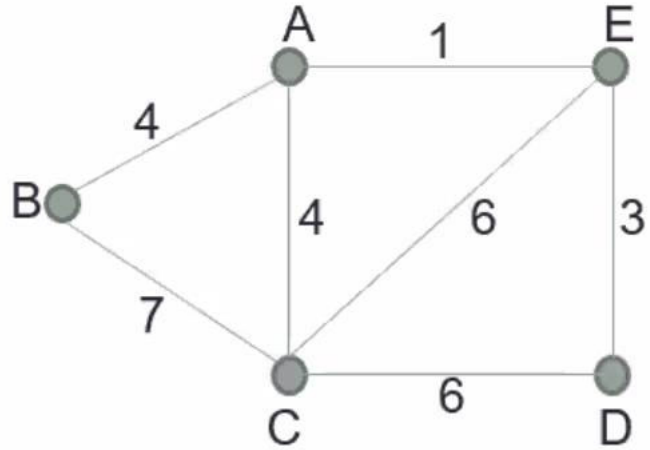


Fig d: Weighted Graph

Directed Graph

- Directed graph: A multigraph M is called a directed graph (digraph) if each edge e in G is assigned a direction, or in other words, each edge e is identified with an ordered pair (u, v) of nodes in G rather than an unordered pair [u, v].

- If G is a directed graph with a directed edge e=(u, v). Then e is also called an arc. The following terminology also used.
  1. e begins at u and ends at v.
  2. u is the origin or initial points of e and v is the destination or terminal points of e.
  3. u is the predecessor of v and v is the successor or neighbor of u.
  4. u is adjacent to v and v is adjacent to u.

Directed Graph Continue…

- Outdegree: The outdegree of a node u in G written as outdeg (u) is the no. of nodes beginning at u.
- Indegree: The indegree of a node u in G written as indeg (u) is the no. of nodes ending at u.
- Source: A node u is called a source if it has positive outdegree but zero indegree.
- Sink: A node u is called a sink if it has positive indegree but zero outdegree.
- Reachable: A node u is said to be reachable from a node v if there is a directed path from v to u.
- Connected or strongly connected: A directed graph G is said to be connected, if for each pair u, v of nodes in G there is path from u to v and there is also a path from v to u.
- Unilaterally Connected: if for any pair u, v of nodes in G there is a path from u to v or a path from v to u.

Directed Graph Continues…

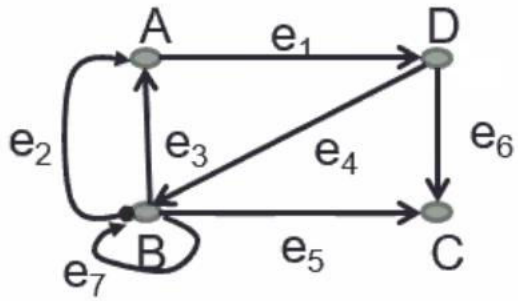


Fig b: Unilaterally Graph

- The edge e_2 and e_3 are said to be parallel, since each begins at B and ends at A.
- The edge e_7 is called a loop, since it begins and ends at the same node B.
- The sequence P_i=(D, C, B, A) is not a path, since (C, B) is not an edge-that is the direction of the edge e_5=(B, C) does not agree with the direction of the path P_i.
- (D, B, A) is a path from D to A, since (D, B) and (B, A) are edges.

Directed Graph Continues…

- Thus A is reachable from D.
- There is no path from C to any other node, so G is not strongly connected. It is unilaterally connected.
- Node C is sink because indeg(C)=2, outdeg(C)=0.
- No node in the Graph are source.
- A directed graph G is said to be simple if G has no parallel edges. Off course it may have loops. But no more than one loop at a given node.
- A nondirected graph G may be viewed as a simple directed graph by assuming that each edge [u, v] represents two directed edges (u, v) and (v, u).

Sequential Representation of Graphs

- Two standard ways to represent graph G in memory.
  1. Sequential Representation, by means of adjacency matrix.
  2. Linked Representation, by means of linked lists of neighbors.

- Adjacency Matrix: Suppose G is simple directed graph with m nodes, and suppose the nodes of G have been ordered and are called $v_1, v_2, \ldots, v_m$. Then the adjacency matrix $A=(a_{ij})$ of the graph G is the m X m matrix defined as follows:

$$a_{ij} = \begin{cases} 1 \text{ if } v_i \text{ is adjacent to } v_j, \text{ that is if there is an edge } (v_i, v_j). \\ 0 \text{ otherwise.} \end{cases}$$

- Such a matrix A, which contains entries of only 1 and 0, is called a bit matrix or a Boolean matrix.

- For undirected graph G, the adjacency matrix is symmetric. This follows from the fact that each undirected edge [u, v] corresponds to the two directed edges (u, v) and (v, u).

- For multigraph, the adjacency matrix of G is the m X m matrix $A= (a_{ij})$ defined by setting a_{ij} equal to the no. of edges from v_i to v_j.

Adjacency Matrix

- For the following directed graph, the adjacency matrix is given by:
- $A=\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$

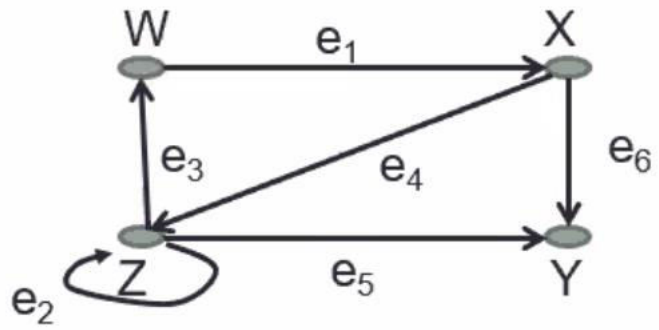


Fig b: Unilaterally Graph

- Note that the no. of 1's in A is equal to the no. of edges in G
- Consider the powers A, A^2, A^3,… of the adjacency matrix A of a graph G. Let $a_k(i, j)$ = the ij entry in the matrix A^k.
- $a_1(i, j) = a_{ij}$ gives the no. of paths of length 1 from node v_i to v_j. Similarly $a_2(i, j)$ gives no. of paths of length 2 from v_i to v_j.

Proposition

- Let A be the adjacency matrix of a Graph G. The $a_k(i, j)$, the ij entry in the matrix A^k, gives the numbers of paths of length k from v_i to v_j.
- Consider the following graph:

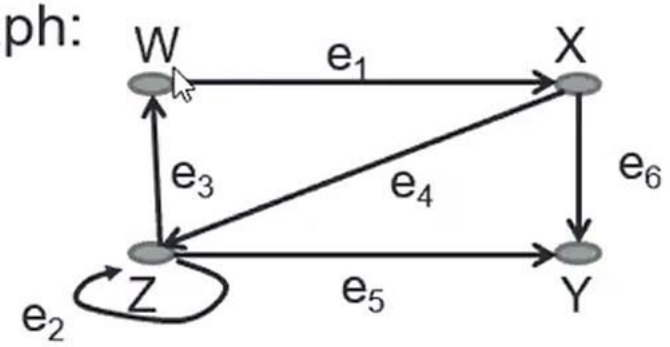


Fig b: Unilaterally Graph

- $A=\begin{pmatrix}0&1&0&0\\0&0&1&1\\0&0&0&0\\1&0&1&1\end{pmatrix}$
- $A^2=\begin{pmatrix}0&0&1&1\\1&0&1&1\\0&0&0&0\\1&1&1&1\end{pmatrix}$ $A^3=\begin{pmatrix}1&0&1&1\\1&1&1&1\\0&0&0&0\\1&1&2&2\end{pmatrix}$ $A^4=\begin{pmatrix}1&1&1&1\\1&1&2&2\\0&0&0&0\\2&1&3&3\end{pmatrix}$

Path of Length k

- In the previous we show the power A^2, A^3, A^4 of matrix A.
- Actually there is a path of length 2 from node w to y, there are two paths of length 3 from node z to y, there are three paths of length 4 from node z to z.
- Suppose $B_r=A+A^2+A^3+\ldots+A^r$. Then the ij entry of the matrix B, gives the number of paths of length r of less from node v_i to v_j.

Path Matrix

- Let G be a simple directed graph with m nodes, v_1, v_2, …,v_m. The path matrix or reachability matrix of G is the m-square matrix $P=(p_{ij})$ defined as follows:

$$P=\begin{cases} 1 \text{ if there is a path from } v_i \text{ to } v_j. \\ 0 \text{ otherwise.} \end{cases}$$

Proposition: Let A be the adjacency matrix and let $P=(p_{ij})$ be the path matrix of a digraph G. Then $p_{ij}=1$ if and only if there is a nonzero number in the ij entry of the matrix

$$B_m=A+A^2+A^3+\ldots+A^m.$$

Proposition

- Consider the graph G, the slide before previous with m=4. Adding the matrix A, A^2, A^3, and A^4, we obtain the following matrix B_4, and by replacing the nonzero entries in B_4 by 1, we obtain the path matrix P of the graph G:

$$B_4=\begin{pmatrix} 2 & 2 & 3 & 3 \\ 3 & 2 & 5 & 5 \\ 0 & 0 & 0 & 0 \\ 5 & 3 & 7 & 7 \end{pmatrix} \quad \text{and} \quad P=\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

- In the matrix P, we see that the node y is not reachable from any of the other nodes.

- Graph G is strongly connected if and if the path matrix P of G has no zero entries.

- So, this particular graph is not strongly connected.

Warshall's Algorithm: Shortest Path

- Warshall gave an algorithm for calculating path matrix efficiently.
- First, we define m-square Boolean matrices $P_0, P_1, \ldots, P_m$ as follows. Let $P_k[i, j]$ denote the ij entry of the matrix P_k. The we define:

$$P_k[i, j] = \begin{cases} 1 & \text{if there is a simple path from } v_i \text{ to } v_j \text{ which does not use any other nodes except possibly } v_1, v_2, \ldots, v_k. \\ 0 & \text{Otherwise.} \end{cases}$$

- In other words,

$P_0[i, j] = 1$ if there is an edge from v_i to v_j.
$P_1[i, j] = 1$ if there is a simple path from v_i to v_j which does not use any other nodes except possibly v_1.
$P_2[i, j] = 1$ if there is a simple path from v_i to v_j which does not use any other nodes except possibly v_1 and v_2.
.....

Accordingly, the elements of the matrix P_k can be obtained by.

$$P_k(i, j) = P_{k-1}[i, j] \vee (P_{k-1}[i, k] \wedge P_{k-1}[k, j]).$$

We can obtain each entry in the matrix P_k by looking at only three entries in the matrix P_{k-1}.

Warshall's Algorithm

- Observe that the matrix P_0=A, the adjacency matrix.
- Since G has only m nodes, the last matrix P_m=P, the path matrix of G.

- Warshall observe that $P_k[i, j]$=1 can occur only if one of the following two cases occurs:

1. There is a simple path from v_i to v_j which does not use any other nodes except possibly $v_1, v_2,\ldots,v_{k-1}$; hence $P_{k-1}[i, j]$=1

2. There is a simple path from v_i to v_k and a simple path from v_k to v_j where each path does not use any other nodes except possibly $v_1, v_2,\ldots,v_{k-1}$; hence $P_{k-1}[i, k]$=1 and $P_{k-1}[k, j]$=1

Warshall's Algorithm

- A directed graph G with M nodes is maintained in memory by its adjacency matrix A. This algorithm finds the (Boolean) path matrix of the graph G.

1. Repeat for I, J=1, 2,…,M:
   If A[I,J]=0, then set P[I,J]=0;
   Else: Set P[I,J]=1.
2. Repeat steps 3 and 4 for K=1, 2,…,M
3. Repeat steps 4 for I=1, 2,…,M
4. Repeat for J=1, 2,…,M
   Set P[I,J]=P[I,J] ∨ (P[I,K] ∧ P[K,J]).

Exit.

Shortest Path Algorithm

- Suppose G is a weighted graph with m nodes, $v_1, v_2, \ldots, v_m$ and each edge e is assigned a nonnegative weight w(e).

- Then G may be maintained in memory by its weight matrix $W=(w_{ij})$, defined as follows:

$$W_{ij} = \begin{cases} w(e) & \text{if there is an edge e from } v_i \text{ to } v_j. \\ 0 & \text{if there is no edge from } v_i \text{ to } v_j. \end{cases}$$

The path matrix P tells us whether or not there are paths between the nodes.

Now we want to find a matrix Q which will tell us the length of the shortest paths between the nodes or, more exactly, a matrix $Q = (q_{ij})$ where

q_{ij} = length of a shortest path from v_i to v_j.

Shortest Path Algorithm Continue…

- Now, we describe a modification of Warshall's algorithm which finds us the matrix Q.
- Now we define a sequence of matrices $Q_0, Q_1, \ldots, Q_m$ whose entries are defined as follows.

$Q_k[i, j]$ = the smaller of the length of the preceding path from v_i to v_j or the sum of the lengths of the preceding paths from v_i to v_k and from v_k to v_j.

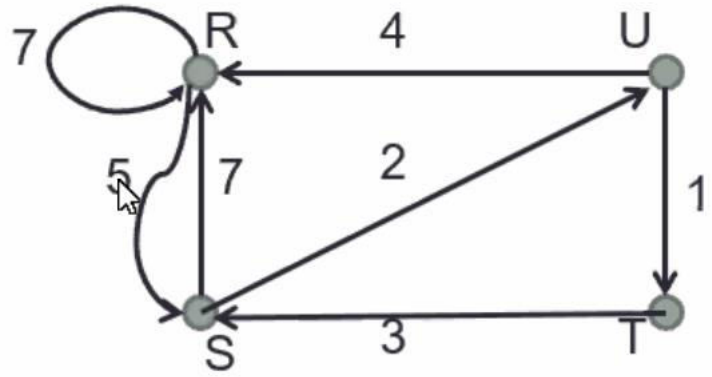
More exactly, $Q_k[i, j] = \text{MIN}(Q_{k-1}[i, j], Q_{k-1}[i, k] + Q_{k-1}[k, j])$

The initial matrix Q_0 is the same as the weight matrix W is replaced by ∞ (or a very, very large number). The final matrix Q_m will be the desired matrix.

Shortest Path Algorithm Continue…

- Consider the weighted graph G with v_1=R, v_2=S, v_3=T, v_4=U.

$$W=\begin{pmatrix} 7 & 5 & 0 & 0 \\ 7 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 1 & 0 \end{pmatrix}$$



- Applying the modified Warshall's algorithm, we obtain the following matrices Q_0, Q_1, Q_2, Q_3 and Q_4=Q. To the right of each matrix Q_k, we show the matrix of paths which correspond to the lengths in the matrix Q_k.

$$Q_0=\begin{pmatrix} 7 & 5 & \infty & \infty \\ 7 & \infty & \infty & 2 \\ \infty & 3 & \infty & \infty \\ 4 & \infty & 1 & \infty \end{pmatrix}$$

$$\begin{pmatrix} RR & RS & - & - \\ SR & - & - & SU \\ - & TS & - & - \\ UR & - & UT & - \end{pmatrix}$$

Shortest Path Length Continue…

$$Q_1=\begin{pmatrix}7&5&\infty&\infty\\7&12&\infty&2\\\infty&3&\infty&\infty\\4&9&1&\infty\end{pmatrix}\quad\begin{pmatrix}RR&RS&--\\SR&SRS&-&SU\\-&TS&--\\UR&URS&UT&-\end{pmatrix}\qquad Q_2=\begin{pmatrix}7&5&\infty&7\\7&12&\infty&2\\10&3&\infty&5\\4&9&1&11\end{pmatrix}\quad\begin{pmatrix}RR&RS&-&RSU\\SR&SRS&-&SU\\TSR&TS&-&TSU\\UR&URS&UT&UTSU\end{pmatrix}$$

$$Q_3=\begin{pmatrix}7&5&\infty&7\\7&12&\infty&2\\10&3&\infty&5\\4&4&1&6\end{pmatrix}\quad\begin{pmatrix}RR&RS&-&RSU\\SR&SRS&-&SU\\TSR&TS&-&TSU\\UR&UTS&UT&UTSU\end{pmatrix}\qquad Q_4=\begin{pmatrix}7&5&8&7\\7&11&3&2\\9&3&6&5\\4&4&1&6\end{pmatrix}\quad\begin{pmatrix}RR&RS&RSUT&RSU\\SR&SURS&SUT&SU\\TSUR&TS&TSUT&TSU\\UR&UTS&UT&UTSU\end{pmatrix}$$

We indicate how the red entries are calculated.

$$Q_1[4,2]=MIN(Q_0[4,2],Q_0[4,1]+Q_0[1,2])=MIN(\infty,4+5)=9$$
$$Q_2[1,3]=MIN(Q_1[1,3],Q_1[1,2]+Q_1[2,3])=MIN(\infty,5+\infty)=\infty$$
$$Q_3[4,2]=MIN(Q_2[4,2],Q_2[4,3]+Q_2[3,2])=MIN(9,3+1)=4$$
$$Q_4[3,1]=MIN(Q_3[3,1],Q_3[3,4]+Q_3[4,1])=MIN(10,5+4)=9$$

Shortest Path Algorithm Continue…

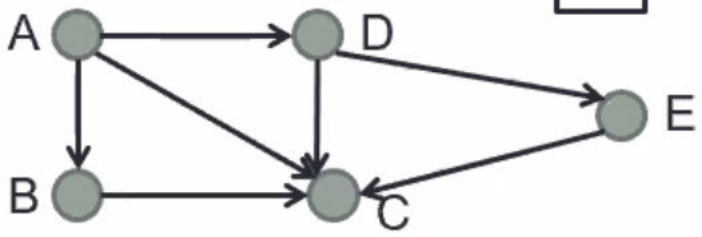
- A weighted graph G with M nodes is maintained in memory by its weight matrix W. This algorithm finds a matrix Q such that Q[I,J] is the length of a shortest path from node V_i to node V_j. INFINITY is a very large number, and MIN is the minimum value function.

1. Repeat for I, J=1, 2,…,M
   If W[I,J]=0, then: Set Q[I,J]:= INFINITY;
   Else: Set Q[I,J]=W[I,J]
2. Repeat Steps 3 and 4 for k=1, 2,…,M
3. Repeat Steps 4 for I=1, 2,…,M
4. Repeat for J=1, 2,…,M
   Set Q[I,J] = MIN(Q[I,J], Q[I,K]+Q[K,J])
5. Exit

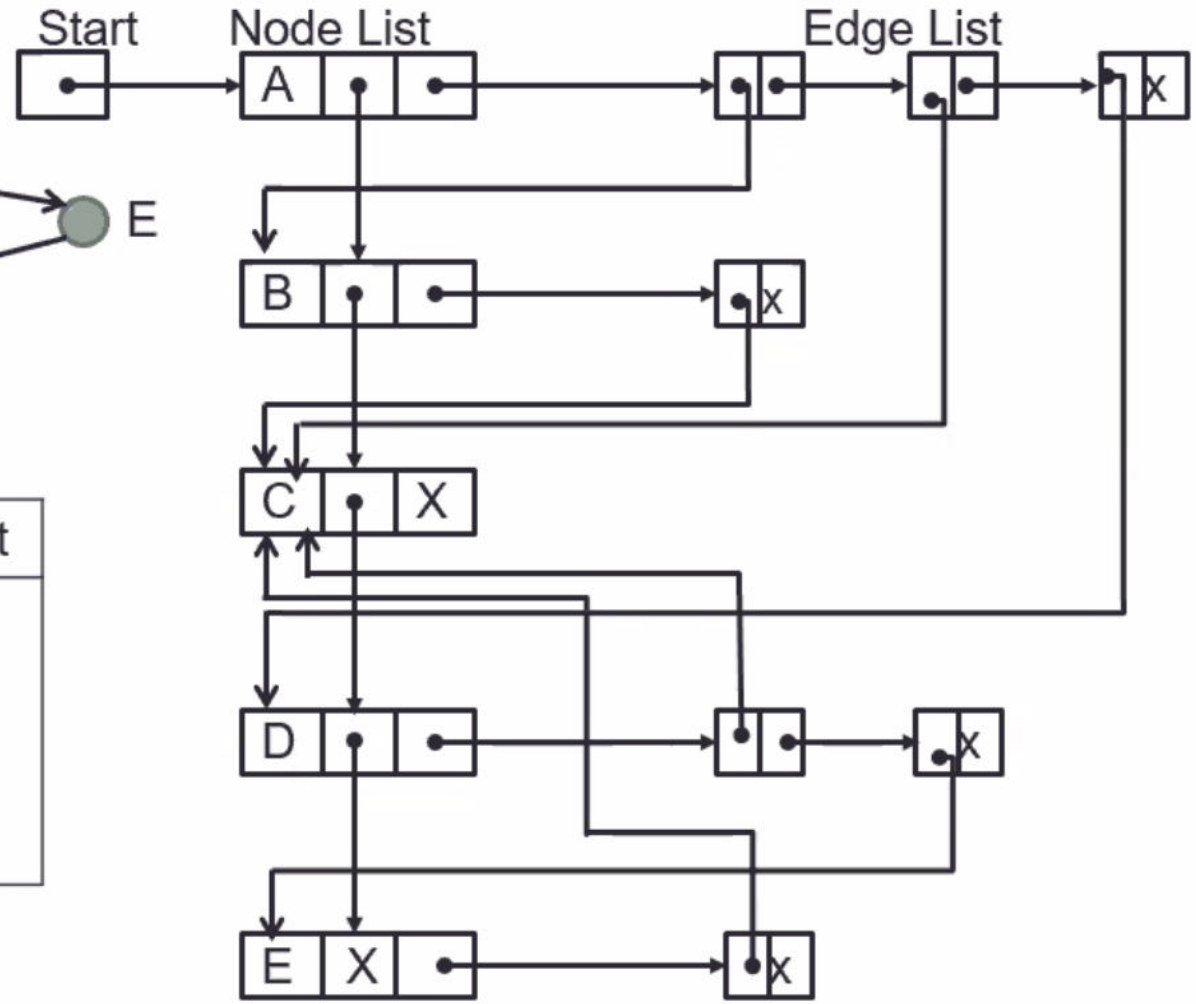
Linked Representation of a Graph

- In sequential representation, there are some drawbacks.
  1. It may be difficult to delete and insert nodes in G. Because the size of A may need to be changed and the nodes may need to be reordered, so there may be many, many changes in the matrix A.
  2. If the number of edges is O(m) or O(m $\log_2$m), then the matrix A will be sparse; hence a great deal of space will be wasted.
- In linked representation, each node in G is followed by its adjacency list, which is its list of adjacent nodes, also called its successors or neighbors. See graph G and its corresponding table on the next slide.

Figure of G and Adjacency list

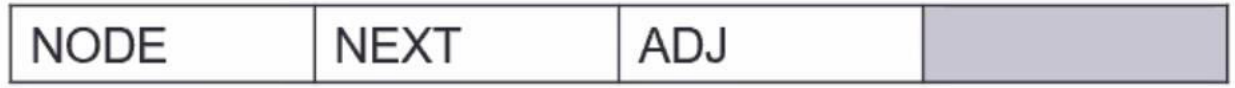


Node	Adjacency List
A	B, C, D
B	C
C	
D	C, E
E	C



Linked Representation Continue…

- Linked representation will contain two lists: a node list NODE and an edge list EDGE.
- Node list: Each element in the list NODE will correspond to a node in G and will be a record of the form.



  - Where NODE will be the name or key value of the node.
  - NEXT will be a pointer to the next node in the list NODE. And
  - ADJ will be a pointer to the first element in the adjacency list of the node, which is maintained in the list EDGE.
  - The shaded area indicates that there may be other information in the record, such as the INDEG of the node, the OUTDEG of the node, the status of the node during execution of an algorithm.

Linked Representation Continue…

- Edge list: Each element in the list EDGE will correspond to an edge of G and will be a record of the form.

  - DEST will point to the location in the list NODE of the destination or terminal node of the edge.
  - LINK will link together the edges with the same initial node, that is, the nodes in the same adjacency list.
  - The shaded area indicates other fields if any(such as edge label, edge weight and so on).

Traversing a Graph

- There are two ways:
  1. Breadth-first search which use a queue as an auxiliary structure to hold nodes for future processing.
  2. Depth-first search which use a stack.
- During the execution of our algorithms, each node N of G will be in one of three states, called the status of N, as follows:
  1. STATUS=1(Ready state.) The initial state of the node N.
  2. STATUS=2(Waiting state.) The node N is on the queue or stack, waiting to be processed.
  3. STATUS=3(Processed state.) The node N has been processed.

Breadth-First Search

- First we examine the starting node A.
- Then we examine all the neighbors of A.
- Then we examine all the neighbors of the neighbors of A.
- And. So. On.

- To guarantee that no node is processed twice, we use a queue that hold those nodes that are waiting to be processed and a field STATUS to tells the current status of any node.

Algorithm:

1. Initialize all nodes to the ready state (STATUS=1)
2. Put the stating node A in queue and change its status to the waiting state (STATUS=2).

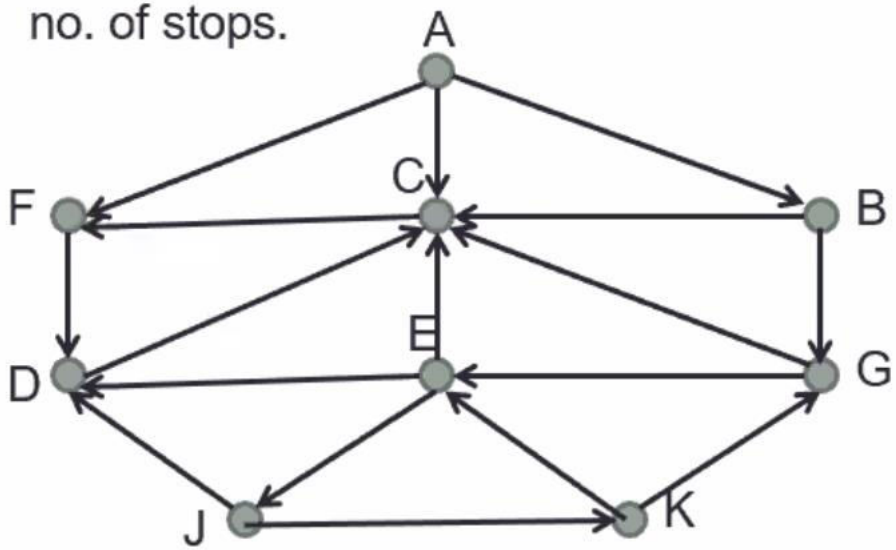
3. Repeat steps 4 and 5 until queue is empty:
4. Remove the front node N of queue. Process N and change its status to the processed state (STATUS=3).

5. Add to the rear of the queue all the neighbors of N that are in the ready state (STATUS=1), and change their status to the waiting state (STATUS=2)
6. Exit.

Breadth-First Search Example

- The algorithm on the previous slide process only those node that are reachable from the stating node A.
- If some nodes are unreachable, then we have to start with another node that is still in the ready state.

- Consider the following graph G that represents the daily flights between cities of some airline, and suppose we want to fly from city A to city J with minimum no. of stops.



Adjacency List	
A	F, C, B
B	C, G
C	F
D	C
E	C, D, J
F	D
G	C, E
J	D, K
K	E, G

Breadth-First Search Cont…

Adjacency List	
A	F, C, B
B	C, G
C	F
D	C
E	C, D, J
F	D
G	C, E
J	D, K
K	E, G

a) Initially add A to Queue and null to Orig as follows:

Front = 1 Queue: A

Rear = 1 Origin: Φ

b) Remove the front element A from Queue by setting Front:=Front+1, and add to Queue the neighbors of A as follows:

Front = 2 Queue: A,F, C, B

Rear = 4 Origin: Φ,A, A, A

The Origin A of each of the three edges is added to Origin.

c) Remove the front element F from Queue by setting Front:=Front+1, and add to Queue the neighbors of F as follows:

Front = 3 Queue: A,F, C, B, D

Rear = 5 Origin: Φ,A, A, A, F

Breadth-First Search Cont…

Adjacency List	
A	F, C, B
B	C, G
C	F
D	C
E	C, D, J
F	D
G	C, E
J	D, K
K	E, G

d) Remove the front element C from Queue by setting Front:=Front+1, and add to Queue the neighbors of C (which are not in the ready state) as follows:

Front = 4 Queue: A,F, C, B, D
Rear = 5 Origin: Φ,A, A, A, F

Neighbor F of C is not added to the Queue, since F is not in the ready state.

e) Remove the front element B from Queue by setting Front:=Front+1, and add to Queue the neighbors of B as follows:

Front = 5 Queue: A, F, C, B, D, G
Rear = 6 Origin: Φ, A, A, A, F, B

Only G is added to Queue, since the other neighbor C is not in the ready state.

f) Remove the front element D from Queue by setting Front:=Front+1, and add to Queue the neighbors of D as follows:

Front = 6 Queue: A, F, C, B, D, G
Rear = 6 Origin: Φ, A, A, A, F, B

Breadth-First Search Cont…

Adjacency List	
A	F, C, B
B	C, G
C	F
D	C
E	C, D, J
F	D
G	C, E
J	D, K
K	E, G

g) Remove the front element G from Queue by setting Front:=Front+1, and add to Queue the neighbors of G as follows:

Front = 7 Queue: A, F, C, B, D, G, E

Rear = 7 Origin: Φ, A, A, A, F, B, G

e) Remove the front element E from Queue by setting Front:=Front+1, and add to Queue the neighbors of E as follows:

Front = 8 Queue: A, F, C, B, D, G, E, J

Rear = 8 Origin: Φ, A, A, A, F, B, G, E

We stop as soon as we J is added to Queue, since J is our final destination.

We now backtrack from J using array Origin to find the path P.

Thus J←E ← G ← B ← A

Depth-First Search

- First we examine the starting node A.
- Then we examine each node N along a path P which begins at A; that is we process a neighbor of A, then a neighbor of a neighbor of A, and so on.

- After coming to a dead end, that is to the end of the path P, we backtrack on P until we can continue along another path P`.
- And so on.

- It is very similar to Breadth-First Search except now we use a stack instead of the Queue.

Algorithm:

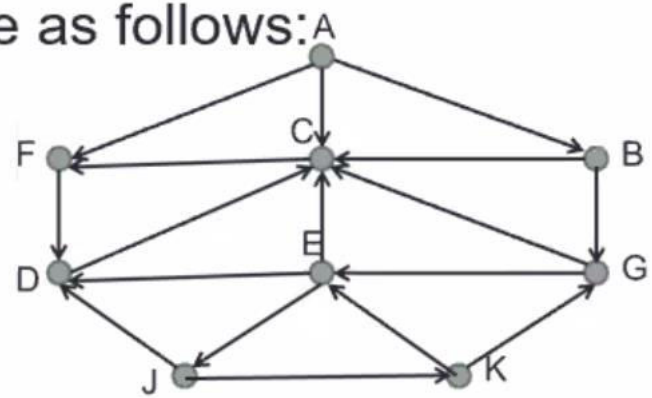
1. Initialize all nodes to the ready state (STATUS=1)
2. Push the stating node A onto stack and change its status to the waiting state (STATUS=2).

3. Repeat steps 4 and 5 until stack is empty:
4. Pop the top node N of stack. Process N and change its status to the processed state (STATUS=3).

5. Push onto stack all the neighbors of N that are still in the ready state (STATUS=1), and change their status to the waiting state (STATUS=2)
6. Exit.

Depth-First Search Example

- Consider the graph on slide 31. Suppose we want to find all the nodes reachable from the node J (including J itself). One way is to use a depth-first search of G starting at the node J. The steps are as follows:



Adjacency List	
A	F, C, B
B	C, G
C	F
D	C
E	C, D, J
F	D
G	C, E
J	D, K
K	E, G

a) Initially push J onto stack as follows:

Stack: J

b) Pop and print the top element J, and then push onto stack all the neighbors of J (those that are in the ready state).

Print: J Stack: D, K

Depth-First Search Example

Adjacency List	
A	F, C, B
B	C, G
C	F
D	C
E	C, D, J
F	D
G	C, E
J	D, K
K	E, G

c) Pop and print the top element K, and then push onto stack all the neighbors of K (those that are in the ready state).

Print: K Stack: D, E, G

d) Pop and print the top element G, and then push onto stack all the neighbors of G (those that are in the ready state).

Print: G Stack: D, E, C

Two neighbors C and E but only C is pushed because E is already there.

e) Pop and print the top element C, and then push onto stack all the neighbors of C (those that are in the ready state).

Print: C Stack: D, E, F

Depth-First Search Cont…

Adjacency List	
A	F, C, B
B	C, G
C	F
D	C
E	C, D, J
F	D
G	C, E
J	D, K
K	E, G

f) Pop and print the top element F, and then push onto stack all the neighbors of F (those that are in the ready state).

Print: F Stack: D, E

g) Pop and print the top element E, and then push onto stack all the neighbors of E (those that are in the ready state).

Print: E Stack: D

None of them are in ready state.

h) Pop and print the top element D, and then push onto stack all the neighbors of D (those that are in the ready state).

Print: D Stack:

The stack is now empty, so the depth-first search of G starting at J is now complete.

The node that were printed J, K, G, C, F, E, D are those which are reachable from J